University of Alberta

Error Identification and Data Recovery in MISR-based Data Compaction

by

Wes A. Tutak

©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 1997

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-22683-2

Canada

University of Alberta

Library Release Form

Name of Author: Wes A. Tutak

Title of Thesis: Error Identification and Data Recovery in MISR-based Data Compaction

Degree: Master of Science

Year this Degree Granted: 1997

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Wes A. Tutak
16207 56 Street
Edmonton, AB
Canada, T5Y 2V1

Date: Aug. 28, 1997.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Error Identification and Data Recovery in MISR-based Data Compaction** submitted by Wes A. Tutak in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Xiaoling Sun

Dr. Bruce Cockburn

Dr. Jonathan Schaeffer

Date: Aug. 28, 1997

Abstract

This thesis presents a new data recovery scheme for use in fault diagnosis in a STUMPS-based testing environment. The proposed scheme transfers partially compacted data from the CUT to the tester and uses analytical methods off-line to recover the information lost during signature compaction. Our proposed data recovery scheme significantly reduces the amount of data transferred from the CUT to the tester and can reduce the tester time used for data retrieval.

Two alternatives to the primary data recovery scheme are presented. The first alternative enhances the error identification resolutions by increasing the amount of partially compacted information obtained from the CUT. The second alternative significantly reduces the testing time and tester complexity by eliminating intermediate signature comparison.

Extensive computer simulations are described that illustrate the merits and feasibility of the new data recovery schemes using pseudorandomly generated circuit responses and ISCAS85 benchmark circuits.

Acknowledgements

I would like to thank my supervisor, Dr. Xiaoling Sun, for her guidance and patience throughout this research project. Her investment of extra time and effort on my behalf is gratefully acknowledged. My thanks also go to the members of my supervisory committee, Dr. Bruce Cockburn and Dr. Jonathan Schaeffer, for their valuable comments. Most importantly, I would like to thank my wife, Darlene, for her support and understanding during the course of my studies.

Contents

1 Introduction **1**

2 Background and Literature Review **8**

2.1 Signature Analysis 8

2.1.1 Serial Data Compaction and Pseudorandom Test Generation . 9

2.1.2 Parallel Data Compaction 10

2.2 The STUMPS Testing Environment 11

2.3 Existing Diagnosis Methods 13

2.3.1 Simulation-based Methods 13

2.3.2 Signature Analysis-based Methods 14

2.3.3 Error Control Code Method 15

2.4 Fault Diagnosis in a STUMPS Environment 16

3 Data Recovery Schemes **20**

3.1 New Data Recovery Scheme 20

3.1.1 Modelling of Parallel Data Compaction 21

3.1.2 Construction of Space Compaction Sequences 22

3.1.3 The Primary Data Recovery Scheme 25

3.2 Definitions and Declarations 26

3.3 Two-known-stream Error Identification Algorithm 30

3.3.1 Algorithm 33

3.3.2 Error Identification Example 37

3.3.3 Complexity Analysis 39

3.4 Two-unknown-stream Error Identification Algorithm 41
3.4.1 Algorithm 43
3.4.2 Error Identification Example 45
3.4.3 Complexity Analysis 46
3.5 Alternative Data Recovery Schemes 47
3.5.1 With Non-overlapping Response Vectors 47
3.5.2 With Simplified Testing Procedure 50

4 Experimental Results **52**
4.1 System Overview 52
4.2 Data Recovery in Pseudorandomly Generated Response Vectors . . . 54
4.2.1 Simulation Environment 55
4.2.2 Results for Overlapping Vectors 56
4.2.3 Results for Non-overlapping Vectors 59
4.3 Data Recovery in Benchmark Circuits 68
4.3.1 Simulation Environment 68
4.3.2 Results 70
4.4 Software Implementation 71

5 Conclusion **78**

Bibliography **82**

Appendices **85**

A DR User Reference Manual **85**

List of Figures

1.1 General testing environment 3

1.2 Signature analysis testing environment 4

2.1 An LFSR with $P(x) = x^3 + x^2 + 1$ 9

2.2 A MISR with $P(x) = x^3 + x^2 + 1$ 10

2.3 The STUMPS architecture 12

3.1 Modified STUMPS architecture 21

3.2 Modelling MISR as concurrent compactions 22

3.3 Data construction system 23

3.4 Data recovery scheme 25

4.1 Software system block diagram 53

4.2 Graphical results for vector size 1024 × 16, 1 block 58

4.3 Node object 73

4.4 Gate class hierarchy 73

4.5 Gate object 73

4.6 A sample SparseMatrix $A[][]$ 76

4.7 The DynamicArray structure 77

List of Tables

3.1 Left shifting MISR space compaction 28
3.2 Right shifting MISR space compaction 29
3.3 Equations derived for vector ($v = 1$) 38
3.4 Equations derived for vector ($v = 2$) 39
3.5 Equations derived for vector ($v = 3$) 40
3.6 Error cancellations in the left compaction stream 43
3.7 Error cancellations in the right compaction stream 44
3.8 Non-overlapping left shifting MISR space compaction 48

4.1 Results for vector size 128×16, 1 block (overlapped vectors) 60
4.2 Results for vector size 128×16, 2 blocks (overlapped vectors) 60
4.3 Results for vector size 256×16, 1 block (overlapped vectors) 61
4.4 Results for vector size 256×16, 2 blocks (overlapped vectors) 61
4.5 Results for vector size 512×16, 1 block (overlapped vectors) 62
4.6 Results for vector size 512×16, 2 blocks (overlapped vectors) 62
4.7 Results for vector size 1024×16, 1 block (overlapped vectors) 63
4.8 Results for vector size 1024×16, 2 blocks (overlapped vectors) 63
4.9 Results for vector size 128×16, 1 block (nonoverlapped vectors) . . . 64
4.10 Results for vector size 128×16, 2 blocks (nonoverlapped vectors) . . 64
4.11 Results for vector size 256×16, 1 block (nonoverlapped vectors) . . . 65
4.12 Results for vector size 256×16, 2 blocks (nonoverlapped vectors) . . 65
4.13 Results for vector size 512×16, 1 block (nonoverlapped vectors) . . . 66
4.14 Results for vector size 512×16, 2 blocks (nonoverlapped vectors) . . 66

4.15 Results for vector size 1024×16, 1 block (nonoverlapped vectors) . . 67
4.16 Results for vector size 1024×16, 2 blocks (nonoverlapped vectors) . . 67
4.17 Characteristics of ISCAS85 benchmark circuits 69
4.18 Results of data recovery in standard benchmark circuits 71

Chapter 1

Introduction

The economic problems of digital testing and fault diagnosis are some of the main concerns in designing and manufacturing digital systems. Digital testing seeks to determine whether a system is functioning correctly. Fault diagnosis attempts to locate the failing components or silicon areas responsible for the malfunction if the system has failed a test [2]. With *integrated circuits* (ICs) rapidly increasing in complexity and decreasing in size, fault diagnosis is essential for IC manufacturing. It is used in the early production phase to identify design and process errors in order to effect design corrections and yield improvements. In addition, it is used to analyze devices that subsequently fail in the field to improve quality and minimize future failures. Improvements to the fault diagnosis methods can impact the overall testing and production cost.

Testing can be broadly categorized into *voltage testing* and *parametric testing*. Voltage testing is concerned with the logic values of circuit outputs (voltage levels) generated by input stimuli as compared with the logic values generated by a reference circuit for the same stimuli. Parametric testing is concerned with the measured values of circuit parameters, such as current, propagation delay or power consumption, and whether they fall within predetermined thresholds. This thesis focuses strictly on voltage testing and henceforth, the term *testing* shall mean voltage testing.

Testing can be either *external* or *internal*. External testing relies exclusively on an external tester to supply stimuli to the *circuit under test* (CUT), and to capture and evaluate the circuit responses. Its chief drawbacks are the expense of the com-

plex, high-speed testing equipment (multi-million dollars); and the large volume of data managed by the tester, resulting in long testing times. Internal testing, such as *built-in self-test* (BIST), reduces the need for complex, expensive testing equipment by including testing circuitry on-chip. As circuit packing density doubles every 18 months, an increasing amount of silicon area (normally $< 10\%$) can be used for BIST. By staying on-chip, BIST can proceed at higher internal circuit speeds making effective testing of larger, more dense ICs practical. Internal testing has become an indispensable technique for testing deep sub-micron ICs.

The last distinction between testing methods is *off-line* versus *on-line* testing. A test is characterized as off-line when the CUT must be taken out of normal operation to be tested. On-line testing methods perform board-level or chip-level testing during normal system operation. On-line testing is necessarily internal as a tester cannot be utilized during the normal system operation. Due to design complexity and high cost, on-line testing is mainly found in safety-critical systems.

Figure 1.1 shows a typical off-line testing environment. The testing process in this environment consists of: (1) applying many stimuli to the CUT; (2) capturing the generated circuit responses; and (3) comparing the responses from the CUT to reference good circuit values to render a "pass/fail" judgement. Two basic components are needed for testing, whether external or internal: a mechanism to provide input stimuli to the CUT, and a mechanism to evaluate the generated circuit responses. In external testing these mechanisms are wholly contained within the tester, while in BIST some mechanisms are implemented on-chip.

In an ideal testing environment, every CUT would be exposed to all possible input stimuli during testing, termed *exhaustive* testing. Due to time and storage constraints, exhaustive testing is only practical for small circuits. For example, to exhaustively test a 100-input circuit with a test system able to apply 1 billion test patterns per second would require approximately 4×10^{13} years, i.e. several orders of magnitude greater than the age of the universe. To keep testing costs low, the test of a single IC should be accomplished in mere seconds. A practical solution to this problem is the *random test* [6]. A random test consists of a large, random selection of test patterns used to expose the CUT to a sample of its input space. A truly random selection of test patterns is undesirable for testing purposes since it is not repeatable (unless stored).

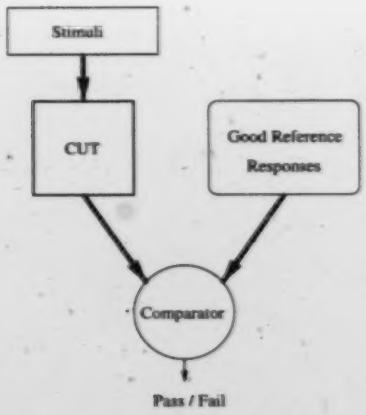

Figure 1.1: General testing environment

Repeatability is necessary to simplify the comparison of circuit responses. Instead, a *pseudorandom* test is used to approximate a random test. It has the properties of a random test but is fully repeatable. A *pseudorandom pattern generator* (PRPG) is a mechanism for generating repeatable sequences of pseudorandom test patterns. PRPGs can be realized directly in hardware for BIST applications.

After applying an input stimulus, randomly generated or otherwise, it is necessary to compare the circuit response with a good reference. The obvious approach would be a direct bit-by-bit comparison of each response with its reference. This is impractical, however, due to the time required to compare numerous circuit responses in the test and the high storage demands of many reference responses. A more practical approach, suitable for BIST, is *data compaction*. Data compaction is destructive data compression with the primary objective of distinguishing different data streams. Data compaction is performed on the sequence of responses generated by the CUT. The final result is termed the *signature* of the CUT. This signature is compared to a precomputed good circuit signature to render a judgment on the CUT. Thus, only a single comparison is performed involving a small quantity of data (typically 16 - 32 bits) permitting short testing times, minimal storage needs, and high test quality. Several data compaction methods exist, including parity checking, transition counting and ones counting [6]. In practice, the mostly commonly used data compaction method is *signature analysis* [6]. Figure 1.2 presents a general signature

analysis-based testing environment.

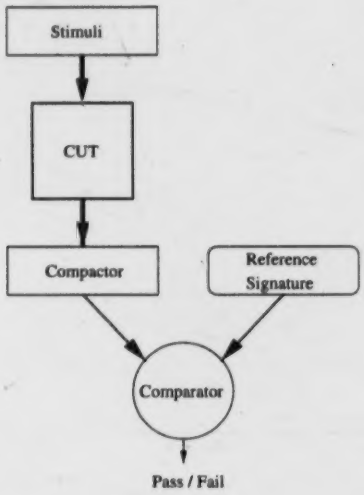

Figure 1.2: Signature analysis testing environment

This thesis addresses issues in BIST employing PRPGs and signature analysis. Specifically, the problem of performing fault diagnosis in this testing environment is investigated. Pass/fail testing is an integral part of IC manufacturing: each fabricated part is placed on a tester for several seconds to determine whether it is functioning correctly. Those ICs that fail the test may require further tester time to gather diagnostic information to identify the defect(s) responsible. Economic constraints only allow mere seconds of tester time per IC. Thus improved methods of performing fault diagnosis can impact the total manufacturing cost of ICs. This thesis explores alternative diagnosis methods for ICs.

When the test of a CUT produces a "fail" result, post-testing diagnosis attempts to identify the physical defect or defects responsible. A *very large scale integrated* (VLSI) circuit may contain millions of transistors. A "fail" in a pass/fail test of an IC does not reveal which of the millions of transistors is defective. Therefore, no design corrections/improvements can be made to eliminate the failure causing problem; hence there is a need for post-testing diagnosis. Diagnosis of a CUT starts from *errors* observed during testing and attempts to identify the *faults* responsible. Errors are circuit responses inconsistent with reference responses. Faults are models

of physical defects within a circuit that may manifest as errors under specific input stimuli. While knowledge of circuit errors reveals the presence of defects in the CUT, knowledge of the faults reveals the location of potential defects in the circuit. This knowledge can be used to correct design problems or make design improvements to enhance fabrication yield.

Signature analysis, while a boon to pass/fail testing, presents unique challenges for fault diagnosis. Diagnosis requires a knowledge of the specific errors produced in the circuit responses. Signature analysis, however, loses this information during compaction toward the final signature of the CUT (typically 16 - 32 bits). Nevertheless, it is a widely used testing method due to advantages in testing speed and ready application in BIST systems. Instead, to fulfill the needs of fault diagnosis, special methods are needed to overcome the extreme data loss and obtain the lost stimulus/faulty response pairs. Ideally, this should be done with restricted built-in hardware, at-speed testing and least possible dependence on an external tester.

STUMPS[1] is a BIST architecture utilizing signature analysis first proposed by Bardell and McAnney [5] (STUMPS is introduced in detail in chapter 2). Although originally proposed for board-level testing, STUMPS has gained in popularity for IC-level testing [12]. It consists of a PRPG to provide test patterns and a parallel compactor to compact internal circuit responses. Test pattern generation and response compaction occur internal to the STUMPS-equipped device. The tester initiates the test and at its completion obtains and evaluates the final signature from the CUT. All circuit responses are lost during compaction and the final signature contains very little information for diagnosis.

A fault diagnosis scheme for use in a STUMPS environment is given by Waicukauski and Lindbloom [20]. This method partitions long circuit response sequences into short blocks (256 test vectors), and compares the intermediate signatures of each block with precomputed fault-free counterparts to identify blocks of responses that contain errors. If a discrepancy occurs, the circuit responses in the faulty block are scanned out in serial and stored for use in fault diagnosis. This method retrieves the uncompacted erroneous circuit responses in the CUT and stores them on the tester. It has

[1]Self-Test Using MISR/Parallel SRSG (shift-register sequence generator)

been shown that often only a few faulty blocks of circuit responses are necessary for successful fault diagnosis.

Following data retrieval, actual fault diagnosis is a two-step process. Structural analysis of the CUT and subsequent fault simulation are used to derive the fault(s) responsible for the observed responses. First, structural analysis of the CUT is performed to create a list of faults that may cause the erroneous circuit responses obtained from the CUT. Each fault in the list is then simulated and the resulting responses are compared with the retrieved responses. If they are equal, the fault is considered as a possible explanation of the observed erroneous behavior, otherwise it is rejected. The final result of the fault diagnosis process is one or more faults modelling potential defects within the CUT.

The objective of this thesis is to explore alternative solutions to the data retrieval method used in Waicukauski's diagnosis scheme [19, 20]. The subsequent fault diagnosis procedures remain unchanged. In our solution, we propose to transfer partially compacted data from the CUT to the tester, and to use analytical methods off-line to recover the information lost during signature compaction.

Our proposed data recovery scheme significantly reduces the amount of data transferred from the CUT to the tester (typically by a factor of 8), and can eliminate the need for decision making by the tester on every intermediate signature. As a result, the tester time used for data retrieval for fault diagnosis can be significantly decreased, while employing a less expensive tester to perform the task. Since a state-of-the-art tester costs millions of dollars, our methods allow significant savings in the overall testing and production cost.

Our simulation results show that we can achieve comparable fault diagnostic resolution as that of Waicukauski. It further validates the feasibility of our proposed data recovery schemes. The remainder of this thesis is organized as follows.

Chapter 2 provides background information and literature review. Data compaction theory is discussed as a basis for the proposed data recovery scheme. The STUMPS testing architecture is then presented in detail. Past diagnosis efforts are reviewed. And lastly, fault diagnosis in the STUMPS environment is treated separately.

Chapter 3 presents the new data recovery scheme. Two error identification algo-

rithms used in the scheme are explained and stated with detailed examples of their operation. Complexity analysis of the two algorithms is given. Lastly, alternatives of the primary data recovery scheme are proposed with their advantages and disadvantages.

Chapter 4 describes the experiments performed to test the performance and feasibility of the data recovery schemes. The details of the simulation environments and justifications are given. The experimental results are presented along with analysis. Lastly, the software system used to perform the simulations is described.

Chapter 5 draws conclusions about the achieved results and summarizes the thesis.

Chapter 2

Background and Literature Review

This chapter presents the background topics relating to testing and diagnosis. Section 2.1 reviews the signature analysis-based data compaction technique. Section 2.2 discusses the STUMPS testing architecture, which is the basis of our new data recovery schemes. Section 2.3 presents a literature review of existing fault diagnosis methods, including simulation-based methods, signature analysis-based methods, and the recent error control code method. Lastly, section 2.4 presents the current STUMPS fault diagnosis scheme.

2.1 Signature Analysis

As introduced in chapter 1, signature analysis is a widely used data compaction technique for the purposes of testing complex VLSI circuits [2]. It transforms the generated response of a CUT into a compact signature (typically 16 - 32 bits). The evaluation of the CUT is then simply the comparison of the obtained signature with a precomputed reference signature; if the two are equal the CUT is judged "good", otherwise it is judged "faulty". The compaction process can be automated and realized in hardware as a *linear feedback shift register* (LFSR) to perform serial compaction or a *multiple-input shift register* (MISR) to perform parallel compaction. This section deals in detail with the operation of these two compactors.

2.1.1 Serial Data Compaction and Pseudorandom Test Generation

Signature analysis is based on the concept of cyclic redundancy checking (CRC), and is implemented in hardware with an LFSR [6]. A shift register is a collection of connected storage elements such that the state of each element is shifted to the next in response to a clock signal. An LFSR is a shift register with a linear feedback network of XOR gates feeding certain storage elements. The next state of a storage element is determined by its source network of XOR gates. To function as a data compactor, the input of the LFSR is supplied the response of the CUT in serial. This perturbs the state of the LFSR according to the feedback network. The signature is the final state of the LFSR following the completion of the test.

The data compaction process performed by an LFSR is equivalent to polynomial division over GF(2) [14]. A binary bit stream can be represented as a polynomial by associating each bit of the stream with the coefficient of a unique power of a dummy variable of the polynomial. For example, the bit stream $b_n b_{n-1} \cdots b_0$ is represented by the polynomial $b_n x^n + b_{n-1} x^{n-1} + \cdots + b_0 x^0$. The LFSR itself can be represented by a polynomial, $P(x)$, with the feedback connections indicated by non-zero terms of the polynomial [6]. Figure 2.1 shows a sample LFSR with polynomial $P(x) = x^3 + x^2 + 1$.

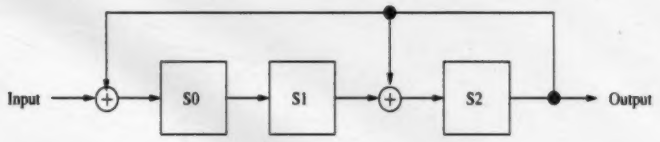

Figure 2.1: An LFSR with $P(x) = x^3 + x^2 + 1$

The compaction process viewed as polynomial division is as follows. The serial input sequence to be compacted is the numerator polynomial, $M(x)$; the LFSR performing the division is the denominator or divisor polynomial, $P(x)$; the serial output of the LFSR is the quotient polynomial, $Q(x)$; and the final state of the LFSR is the remainder polynomial or signature, $S(x)$. Mathematically, signature compaction is formulated as:

$$Q(x) = \frac{M(x)}{P(x)} + S(x).$$

To perform the polynomial division, the LFSR is initialized to an all-zero state and the serial input sequence, $M(x)$, is presented to the LFSR input, high-order bit first. The quotient bit sequence, $Q(x)$, is shifted out of the LFSR output, high-order bit first, and the final state of the LFSR following compaction is the signature, $S(x)$.

As well as serving as data compactors, LFSRs can be used for test pattern generation. An LFSR, initialized to a non-zero state, will cycle through a pseudorandom sequence of states with successive applications of a clock signal. The sequence of states is determined by the polynomial of the LFSR. An LFSR that generates the maximum length sequence of non-repeating states is termed a *primitive* LFSR. A primitive m-bit LFSR will generate all states corresponding to the possible combination of m bits but for the all-zero state. The maximum length sequence of states of an m-bit primitive LFSR is thus $(2^m - 1)$ states.

2.1.2 Parallel Data Compaction

A drawback of employing LFSRs for data compaction is the single data input which only permits the compaction of a serial data stream. To compact multiple data streams in parallel, an LFSR can be modified into a MISR [6] by adding an input and an XOR gate between every two adjacent register cells, retaining the same feedback connections as the original LFSR. An m-bit MISR is able to compact m bits of data per clock cycle. Figure 2.2 shows an example MISR with $P(x) = x^3 + x^2 + 1$.

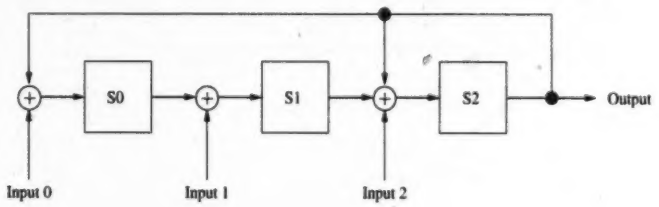

Figure 2.2: A MISR with $P(x) = x^3 + x^2 + 1$

For notational convenience, a polynomial can be represented by a binary sequence where each non-zero coefficient corresponds to a "1", and "0", otherwise. As an example, the polynomial, $P(x) = x^3 + x^2 + 1$, can be represented by the binary sequence, 1101.

2.2 The STUMPS Testing Environment

Many off-line BIST architectures employ signature analysis along with other techniques such as scan testing and partitioning [2] to increase the testability of a circuit. The goal of scan testing is to transform a sequential circuit into a purely combinational circuit for testing purposes. The response of the circuit is then solely dependent on the current stimulus, not past stimuli. To achieve this goal requires access to the internal circuit storage elements where normally no such outside access exists. To that end, scan testing replaces all memory elements with special scan registers connected to form scan chains. A scan register is a dual-mode memory element: in normal mode data is loaded from the data input line; in scan-mode data is loaded from a separate scan-in port. The scan-in and scan-out ports of a scan chain are made externally accessible, permitting data to be serially scanned in or out of the internal memory elements. The other testing technique of circuit partitioning attempts to divide a circuit into many sub-circuits that can be tested separately. Partitioning can be realized with scan testing by creating many separate scan chains to divide a circuit.

An off-line BIST architecture incorporating scan testing and partitioning is the *self-test using MISR/parallel SRSG*[1] (STUMPS) testing architecture [5]. It was originally proposed to test multichip modules at the board-level. A special testing chip implements the SRSG and MISR components of STUMPS which, respectively, generate test patterns for the other chips on the board and compact in parallel their output responses.

Each chip to be tested must utilize scan-based flip-flops [5], configured into scan chains (or data streams). A number of scan chains, per board, are formed by directly connecting the scan-in and scan-out ports of individual chips. The scan chains are supplied pseudorandom test patterns in parallel from the SRSG. By scanning in known test patterns into the scan chains, a sequential circuit is converted into a combinational circuit during testing. Combinational circuits are easier to test as their response depends only upon the current input vector, not on past inputs.

The scan chains provide the inputs to the combinational logic blocks and capture the generated responses. Normal circuit operation is governed by a system clock or

[1] Acronym of shift-register sequence generator.

clocks. Scan testing also introduces a separate test clock to govern the serial flow of data within scan chains. With multiple applications of the test clock, data from the SRSG is scanned into the scan chains, loading test patterns into the chips to be tested. The regular system clock is then asserted once to capture the responses from the chips back into the scan chains. The subsequent test patterns are then scanned in, while simultaneously, the circuit responses are being scanned out to the test chip where they are compacted by the MISR. After the application of many test patterns, the final signature is scanned out of the test chip and compared with a error-free signature to determine whether response errors were detected.

STUMPS has since become a standard IC-level BIST architecture [12]. Memory elements are realized as scan registers connected to form scan chains. The functions of the test chip are implemented directly within the IC as dedicated BIST resources. Figure 2.3 shows the IC-level STUMPS architecture showing the configuration of scan chains, SRSG and MISR.

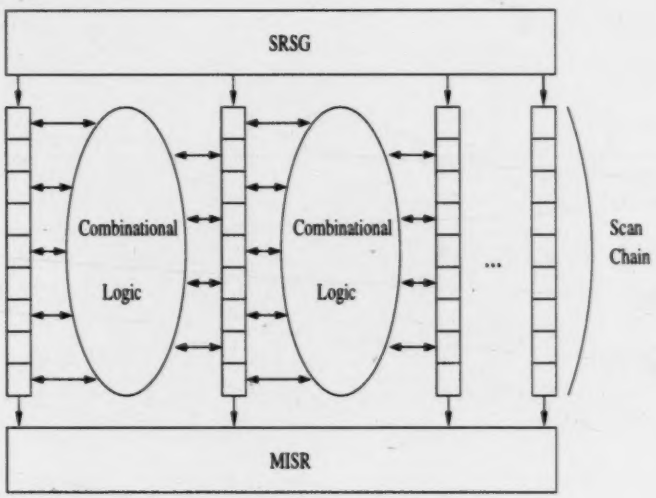

Figure 2.3: The STUMPS architecture

2.3 Existing Diagnosis Methods

Testing and diagnosis have different purposes and conflicting requirements. Testing should be fast, comprehensive and inexpensive since it is performed on every device manufactured. Techniques used to speed testing, such as signature analysis, severely hamper fault diagnosis by compacting, and losing, the circuit responses necessary for diagnosis. Diagnosis is only performed on devices that fail testing, thus the primary requirements are of diagnostic accuracy and detail. The results of diagnosis must be accurate and specific to focus attention on the location of the defect so that it can be quickly corrected. To reduce cost, it is desirable to perform diagnosis with existing testing methods and with minimal tester support.

A circuit may have countless possible physical defects that can produce faulty behavior. Diagnosis employs fault models to systematically characterize the majority of the potential defects. A fault model is a set of rules and assumptions which describe the effects that defects have on digital circuits [2]. The results of fault diagnosis are one or more faults (or fault classes) in the adopted fault model. Many fault models have been proposed to describe various defects, including: briding faults, transition faults, delay faults, and stuck-open faults. The most common is the *stuck-at* fault model, it assumes fault-free logic gates with defects causing signal lines between gates to be permanently stuck at either logic 1 or logic 0, regardless of stimulus [2]. Much of the early work in fault diagnosis has been based on the assumption of the stuck-at fault model. This section reviews published literature as a history of fault diagnosis.

2.3.1 Simulation-based Methods

Early diagnosis approaches attempt to enumerate the behavior of faults in the assumed fault model [1]. A fault dictionary is compiled by simulating the CUT with every fault and recording the corresponding circuit response. Diagnosis then consists of the simple task of locating the observed response of the CUT in the fault dictionary and noting the corresponding fault(s). Disadvantages of this technique are set by the limits of the adopted fault model and the inability to diagnose multiple faults.

2.3.2 Signature Analysis-based Methods

Signature analysis based methods attempt to solve the problem of response data loss inherent with data compaction. Considerable effort has been made in identifying and locating errors in the uncompacted circuit responses based on the observed signature [6, 8, 9, 11, 13, 19, 20, 21]. There are three types of signature analysis-based fault location techniques: fault dictionary, algebraic analysis and intermediate signatures.

Fault Dictionary

The fault dictionary-based method constructs a look-up table containing the modeled faults and their corresponding faulty signatures [6]. Diagnosis consists of locating the observed faulty signature in the dictionary. The small size of a signature makes this method limited at discriminating individual faults as many faults may produce the same signature.

Algebraic Analysis

Algebraic analysis methods attempt to compute the erroneous circuit response from the faulty signature obtained from testing.

The first such method was given by McAnney and Savir [11]. It uses an LFSR that is the reciprocal of the LFSR used for compaction. The reciprocal LFSR is initialized with the faulty signature obtained from the CUT then clocked to reverse the compaction and compute where the error was introduced. This method constrains the test length to be no greater than the state space of the LFSR. At most two errors in the test sequence can then be identified.

A similar method by Chan and Abraham [8] is applicable to both serial and parallel compactors. It uses state transition matrices to describe the compaction process. An analytical formulation is given to calculate the error location in the test sequence. The method is limited to identifying which compaction step introduced the errors or the channel(s) containing the errors.

A method employing a number of cyclic registers to perform compaction was presented by Savir and McAnney [13]. Utilizing multiple cycling registers, the errors in the test sequence are computed from the signatures remaining in the registers. The

number of errors that can be effectively identified is approximately half the length of the shortest register. Three cycling registers of relatively prime lengths (i.e. sharing no common divisors) can identify up to 50 errors in a sequence of 2^{20} bits using a 300-bit signature. Disadvantages include the physical size of the long cycling registers and the added storage for a fault dictionary of 300-bit signatures.

In [9], a partitioning scheme to decompose a final error signature into partial error signatures is presented. Existing methods are then used to derive the error sequence for each partial error signature. The final error sequence is given as the product of the partial error sequences. Although this scheme can identify multiple errors, a particular disadvantage is the time and space complexity of the partitioning algorithm for significant numbers of errors.

Intermediate Signatures

Intermediate signature methods are based on signatures obtained at regular intervals during testing. For diagnostic purposes, circuit responses are partitioned into short blocks. An intermediate signature is obtained after the compaction of each block of responses. The intermediate signatures are compared with error-free counterparts to target failing blocks for diagnosis.

Waicukauski's intermediate signature method [19, 20] is described in chapter 1 on page 5. Intermediate signatures are computed every block of 256 test vectors. Blocks with faulty signatures are targeted for data retrieval and subsequent fault diagnosis. This method is treated further in section 2.4.

An alternative approach is employed at Northern Telecom [21]. In this method intermediate signatures are computed after every test vector. Data retrieval is not performed. Instead, fault simulation is used to locate the fault or faults that produce the observed intermediate signatures.

2.3.3 Error Control Code Method

The diagnosis method in [22] uses a special programmable MISR (PMISR) to perform compaction. A set of equation based on Reed-Solomon codes [10] are obtained from the faulty signatures and solved to identify the error-capturing frames. Each scan

chain is then retested to locate the actual erroneous flip-flops. Drawbacks of this method include the need to retest the CUT with different configurations of the PMISR and the computational expense necessary to solve the system of equations.

2.4 Fault Diagnosis in a STUMPS Environment

Fault diagnosis is an important process in the design of IC chips with sub-micron technologies. As well as helping to correct design errors, it can be used to improve yield and increase circuit reliability. Fault diagnosis is the opposite of fault simulation: it starts with a stimulus and observed faulty circuit response, and then determines the fault set that can produce the faulty response given the same stimulus. To reduce the fault set, many stimulus/faulty response pairs must be considered. Knowledge of the fault(s) potentially responsible for the observed behavior is then used by the designer to correct design problems or improve yield.

Fault diagnosis requires that many stimulus/faulty response pairs be obtained from a CUT. However, this is at odds with current internal testing methods. The necessity of testing increasingly dense ICs has led to the innovation of BIST. One form of BIST is STUMPS which is both an aid and a hindrance to fault diagnosis. It is an aid since it permits the observation of many internal circuit nodes, aiding diagnostic resolution. However, it is a hindrance, as during testing these observation points are inaccessible from off-chip. Additional, post-testing time must be expended to scan out this internal circuit information where it can be used for fault diagnosis.

Once initialized, the testing of a STUMPS equipped IC occurs mainly on-chip. The result obtained is a final signature (typically 16 - 32 bits long) which is the compacted response of the CUT for the entire test set. This signature is then compared off-chip by the tester with the good signature (obtained through logic simulation of the good circuit) to produce a pass/fail judgment of the IC. By staying mainly on-chip, testing can proceed at much higher speeds.

The final signature is suitable for rendering a pass/fail judgment of a CUT, however on its own it is grossly inadequate for fault diagnosis. Consider a STUMPS implementation consisting of 16 data streams, each of 1024 bits and a test length of 100,000 patterns. Each circuit response consists of (16×1024) bits or 2 Kbytes

of information; while the entire test consists of 200 Mbytes of information. The circuit response information lost during data compaction is unrecoverable from a final signature of several bytes. This makes IC-level fault diagnosis a difficult and costly task.

Data retrieval is one method for obtaining the lost circuit response information [20]. Recall that in STUMPS, access to all data streams is only available through two ports, a scan-in port for input and a scan-out port for output. Data retrieval in STUMPS is the process of scanning out in serial the contents of all data streams to obtain the response of the CUT.

To perform fault diagnosis, many stimulus/faulty response pairs must be scanned out from the CUT. However, scanning out the entire circuit response for every stimulus in the test set is time-consuming and unnecessary. Not every circuit response will be faulty, certain stimuli may not induce errors in circuit responses, producing the same responses as the good circuit. Fault-free circuit responses can be determined through circuit simulation, thus it is desirable to only scan out the faulty circuit responses.

For the purposes of data retrieval, the test set is divided into discrete intervals of one or more test patterns. Each interval has an *intermediate signature* computed by logic simulation of the good circuit. This is the intermediate result of compacting all responses up to and including the responses from the current testing interval. All such signatures are compiled into a dictionary of intermediate signatures. During data retrieval, the intermediate signatures obtained from the CUT are compared with their counterparts in the dictionary. A discrepancy indicates that the preceding test interval must contain at least one faulty response, whereupon all responses in the present interval are scanned out to be used for fault diagnosis.

Consider a system with intervals consisting of 100 test patterns, the specific steps of the data retrieval process are [6]:

Step 1: Initialize the BIST circuitry in the CUT to the beginning of the test sequence.

Step 2: Apply the next 100 test patterns to the CUT.

Step 3: Scan out the intermediate signature from the CUT.

Step 4: Compare the signature with the counterpart in the dictionary. If the signatures are the same, the state of the BIST circuitry in the CUT is restored to the state before the signature was scanned out. Otherwise, if the signatures differ, the BIST circuitry in the CUT is restored to the state at the start of the current interval. The 100 test patterns are then re-applied, but instead of being compacted each response is scanned out and stored for further fault diagnosis. Once all responses in the interval have been scanned out, the BIST circuitry in the CUT is restored to that of the good circuit at the end of the current test interval.

Step 5: If there are more intervals in the test set, go to **Step 2**.

Step 6: The stored responses are transferred from the tester to a workstation where fault diagnosis can proceed off-line.

After data retrieval is complete, fault diagnosis [20] is performed to determine the fault class(es) responsible for the observed faulty responses. In addition to the faulty responses, the following data and systems are necessary to perform fault diagnosis: (1) a structural description of the CUT, (2) a fault simulator, and (3) a PRPG to generate the test patterns in the test. The first step of fault diagnosis involves the structural analysis of the circuit to create a minimal fault list. Subsequently, each fault in the list is simulated with the generated stimuli and the resulting responses are compared with the retrieved responses. If the simulated responses match all retrieved responses, the fault is accepted. Otherwise, the fault is rejected as it does not reproduce all observed responses. The final result of fault diagnosis is a set of faults that can reproduce the observed faulty behavior and thus are potentially responsible for the defect(s) in the CUT.

As can be seen from the previous steps, data retrieval is a complex, lengthy process as compared with pass/fail testing. A pass/fail judgment of a CUT involves a single, uninterrupted application of the test set followed by the scan-out and comparison of the final signature. The only tester-CUT interaction is at the start to initiate the test, and at the end to scan-out the final signature. Data retrieval, on the other hand, demands many more interactions between the tester and the CUT. Each test

interval is initiated and halted, the intermediate signatures are scanned out, the BIST circuitry is reset, and ultimately the circuit responses are scanned out. Data retrieval occupies an expensive testing system for an extended length of time that can otherwise be used to verify many more newly fabricated ICs. It transfers more data from the CUT than may be necessary to accomplish fault diagnosis.

The data recovery schemes presented in the following chapter attempt to address these shortcomings. The new schemes obtain the faulty circuit responses from partially compacted information transferred from the CUT and exploit analytical methods implemented off-line.

Chapter 3

Data Recovery Schemes

In this chapter we present a new scheme for data recovery in MISR-based data compaction. The primary data recovery scheme [16] is introduced in section 3.1. New terms with examples, are defined in section 3.2. Two new error identification algorithms, central to the data recovery scheme, are presented in sections 3.3 and 3.4, respectively, along with detailed examples of their operation and analysis of their computational complexity. The first algorithm is able to perfectly identify all errors confined to any two known data streams. The second algorithm generalizes this approach to unknown data streams, permitting the identification of errors in any two data streams. Lastly, alternatives to the primary data recovery scheme are proposed in section 3.5. The alternatives enhance the primary scheme by increasing the amount of information available for error identification and by significantly reducing testing time.

3.1 New Data Recovery Scheme

The proposed data recovery scheme consists of both a hardware and software component. The hardware component is an additional BIST resource in the form of a second MISR added to the STUMPS architecture. The software component implements error identification algorithms that compute the erroneous data bits in the uncompacted circuit responses. The erroneous data bits are combined with the good circuit responses obtained by simulation to produce the uncompacted circuit responses. The

new scheme replaces the data retrieval method presented in chapter 2. It requires additional BIST hardware and off-line computation in exchange for the elimination of extra tester-CUT interactions and reduction of the volume of data transferred between tester and CUT.

Recall from chapter 2 that the traditional STUMPS architecture has a single MISR that compacts the circuit response. The necessary hardware support for the data recovery scheme consists of the standard STUMPS architecture with the addition of a left shifting MISR and a corresponding scan-out port, as shown in figure 3.1. With two MISRs, one shifting left and another shifting right, two different *quotient sequences*, Q_L and Q_R, can be obtained from the MISRs. The quotient sequences are the only information obtained from the CUT for data recovery. The proposed data recovery scheme uses analytical techniques to compute the circuit responses lost during the data compaction based on the quotient sequences.

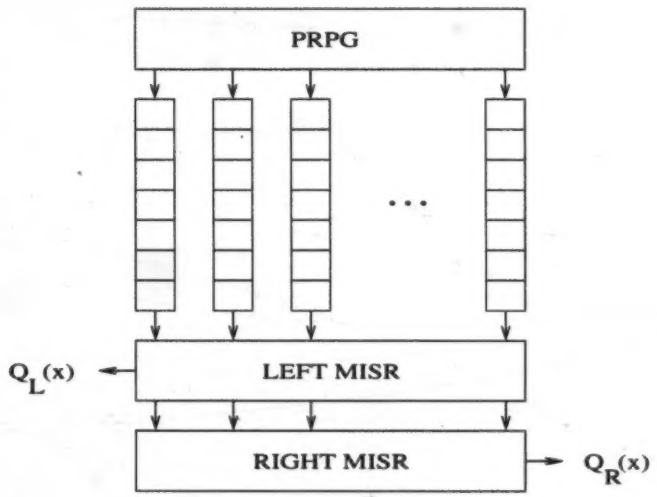

Figure 3.1: Modified STUMPS architecture

3.1.1 Modelling of Parallel Data Compaction

The signature compaction process in a MISR can be considered as two concurrent compaction processes [23]. The first process is *space compaction* which transforms n, m-bit words into a serial stream of $(n + m - 1)$ bits, which is termed the *space*

compaction sequence. The second process is *time compaction* which performs polynomial division of the space-compacted stream by an LFSR with the same feedback polynomial as the MISR. An example of the two processes is shown in figure 3.2. This dual view of MISR-based data compaction permits the construction of the space compaction sequence from the quotient sequence and the signature. The space compaction sequence can then be used to identify errors in the uncompacted circuit response. In figure 3.2 it can be seen that each bit of the space compaction sequence is the modulo-2 sum of at most three response bits. Thus, errors in these response bits are localized to this one bit of the space compaction sequence. No such localization of errors exists in the quotient sequences, however: the first introduced error bit will perturb many subsequent bits in the quotient sequence.

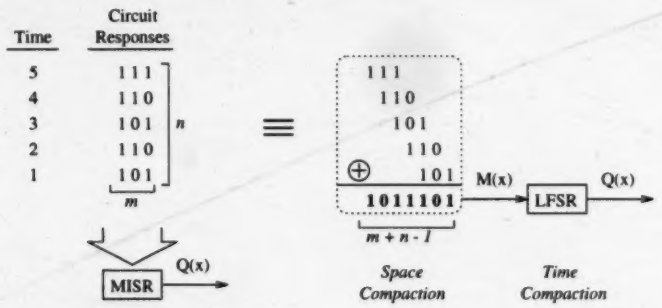

Figure 3.2: Modelling MISR as concurrent compactions

3.1.2 Construction of Space Compaction Sequences

We term the reverse of signature compaction *data construction.* Thus, equation

$$M(x) = P(x)Q(x) + S(x)$$

can be considered as the process of constructing the input sequence, $M(x)$, from the given quotient sequence, $Q(x)$, divisor polynomial, $P(x)$, and signature, $S(x)$.

Consider a data block of $(n \times m)$ bits, where n and m are the numbers of rows and columns of the data block, respectively. Let $M(x)$ be the $(n+m-1)$-bit serial stream space compacted by an m-bit MISR. We assume that the bits which made up $Q(x)$ are collected during the data compaction. By linearity, the process of constructing

$M(x)$ from the given $P(x)$, $Q(x)$ and $S(x)$ can be considered as:

$$M(x) = M_1(x) + M_2(x) + \cdots + M_n(x) + S(x),$$

where $M_k(x) = Q_k(x)P(x)$, and $Q_k(x)$ equals the quotient, $Q(x)$, with all bits except the k-th bit set to 0. During compaction, a $Q(x)$ bit is shifted out of the MISR at each clock cycle. This bit can then be used to build one subsequence, $M_k(x)$, and to complete the construction of one bit of $M(x)$.

Example 3.1 Consider the MISR in figure 2.2, with $P(x)$=1101, and the data block and the space compaction stream, $M(x)$=1011101, in figure 3.2. The $Q(x)$ collected during time compaction is 11000, and the signature, $S(x)$, is 001. According to the partial construction method described above, we have:

$$M_1(x) = 1101000;\ M_2(x) = 0110100;\ M_3(x) = 0000000;\ M_4(x) = 0000000;$$
$$M_5(x) = 0000000, \quad \text{and}$$
$$M(x) = M_1(x) + M_2(x) + M_3(x) + M_4(x) + M_5(x) + S(x) = 1011101.$$

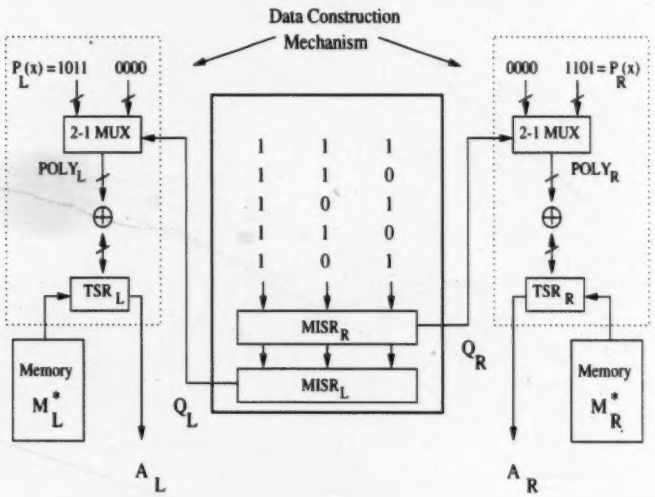

Figure 3.3: Data construction system

Figure 3.3 depicts an implementation of the data construction technique developed by Dr. Sun et al. [15, 17], which can be realized in either hardware or software.

The system consists of the following components. $MISR_R$ and $MISR_L$ implement the divisor polynomial, $P_R(x)$=1101, and its reciprocal, $P_L(x)$ = 1011, respectively. $Q_L(x)$ and $Q_R(x)$ are the quotient sequences obtained during the data compaction in real time. The memory contains the predetermined error-free space compaction sequences, $M_L^*(x)$ and $M_R^*(x)$. TSR_L and TSR_R are $(m+1)$-bit left and right shift registers, respectively. $POLY_L$ and $POLY_R$ are 2-1 MUXs. Q_L and Q_R serve as the MUX *Select* signals to construct $M_L(x)$ and $M_R(x)$, respectively. The binary representations of the polynomials of the MISRs are chosen when *Select*=1; otherwise all zeros are chosen. The output of the system is the compaction sequence error masks,

$$A_L = M_L \oplus M_L^*$$

$$A_R = M_R \oplus M_R^* ,$$

generated in real-time during testing of the CUT. The proposed data recovery system then uses the A_L and A_R sequences to recover the information bits lost during data compaction.

We explain the right shifting operation of the system using the data in figure 3.2, where m=3, n=5, $M_R(x) = M_R^*(x)$=1011101 and $Q_R(x)$=11000. The operation of the left shifting process is symmetric.

1. The most significant $(m+1)$ bits of $M_R^*(x)$, 1011, are loaded into TSR_R.

2. The first 3-bits of the (3×5) circuit response, 101, are fed into and compacted by the two MISRs simultaneously. The first **1** output bit of $Q_R(x)$ from $MISR_R$ causes 1101 to be selected by $POLY_R$. In the case of a **0** output bit of $Q_R(x)$, 0000 is selected.

3. The contents of TSR_R are XORed with that of $POLY_R$ producing 0110, which is then stored back to TSR_R.

4. The contents of TSR_R are shifted 1-bit to the left and 1-bit of the remaining $M_R^*(x)$ in memory is simultaneously shifted into TSR_R. The bit **0** of TSR_R is shifted out. A **0** indicates "no error" in the last bit of the space compaction sequence, and a **1** indicates "an error".

5. Repeat steps 2 to 4 until the end of the data compaction process.

3.1.3 The Primary Data Recovery Scheme

The newly proposed data recovery scheme takes as input the left and right quotient sequences and produces as output the uncompacted circuit responses. Figure 3.4 shows a block diagram of the data recovery scheme. The first step entails the construction of the left and right space compaction sequences, M_L and M_R, from the quotient sequences, Q_L and Q_R, obtained from the CUT. The space compaction sequences with their precomputed error-free counterparts, M_L^* and M_R^*, are used to compute the compaction sequence error masks, A_L and A_R. These are used by the error identification algorithms to compute the errors present in the original uncompacted circuit responses. Finally, the identified errors are combined with the good circuit responses, obtained by simulation, to produce the complete uncompacted circuit responses.

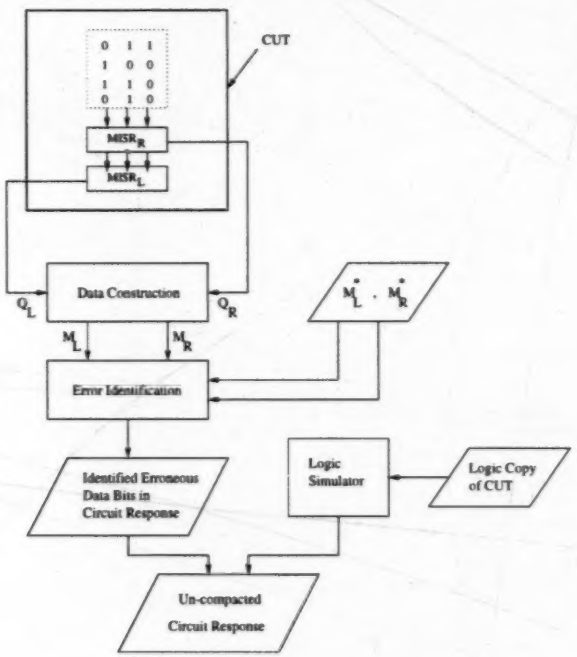

Figure 3.4: Data recovery scheme

The new data recovery process proceeds as follows:

Step 1: Initialize the BIST circuitry in the CUT.

Step 2: Capture the Q_L and Q_R sequences from the left and right shifting MISRs

after a faulty block is identified by the comparison of intermediate signatures during a test.

Step 3: Reconstruct the left and right space compaction sequences, M_L and M_R, from the Q_L and Q_R sequences, respectively.

Step 4: Invoke the analytical error identification algorithms to compute the set of erroneous circuit response bits that produced the M_L and M_R sequences.

Step 5: Combine the results of step 4 with the good circuit responses obtained via logic simulation to obtain the faulty circuit responses.

Step 6: Proceed with further fault diagnosis to locate defective components/areas based on the recovered erroneous circuit responses obtained in step 5.

The following section formally defines the terminology used. The subsequent sections present the error identification algorithms required by the data recovery process (step 4).

3.2 Definitions and Declarations

The basic entity used in the discussion of the error identification algorithms is the *response vector*, which is the response of the CUT to one stimulus. In a STUMPS environment, the response vector can be viewed as an $(n \times m)$ data block, where n is the length of each data stream (assuming all data streams are of equal length), and m is the number of data streams compacted in parallel by the m-bit MISRs. Further, assume that V such response vectors were generated by V test patterns or test vectors.

Our data recovery scheme employs two MISRs during data compaction, one shifting to the left and another shifting to the right. In our model of MISR-based compaction, the circuit responses can be viewed as both a left and right space compaction that produce the left and right space compaction sequences, respectively.

Definition 3.1 A *response vector* is an $(n \times m)$ collection of data bits generated by the CUT from the application of one test pattern.

Definition 3.2 The *left space compaction sequence*, denoted as $M_L(x)$, is the sequence of bits resulting from the space compaction performed by the left shifting MISR. Further, $M_L^*(x)$, denotes the error-free left space compaction sequence obtained by simulation of the good circuit. Analogously, for the right-space compaction, we have $M_R(x)$ and $M_R^*(x)$.

Any data bit in a response vector can be identified by the row and columns indices, i, j respectively, indexing into an $(n \times m)$ data block. A test set will consist of some number V compacted response vectors. Elements are further discriminated according to their respective response vector, v.

Definition 3.3 An *element* is a data bit in one $(n \times m)$ response vector. For a test length of V vectors, any element, written $E_{i,j}^v$, can be uniquely identified by the tuple (v, i, j), where v is the vector $(1 \leq v \leq V)$, i is the data row $(1 \leq i \leq n)$, and j is the data column or data stream $(1 \leq j \leq m)$.

Elements in the left or right compaction are grouped into left and right space compaction columns, respectively. The left space compaction columns of response vector v, are labeled from left to right as L_1^v, L_2^v, ..., L_{n+m-1}^v. Similarly the right space compaction columns of response vector v, are labeled from right to left as R_1^v, R_2^v, ..., R_{n+m-1}^v. As with individual elements, space compaction columns are identified as to their corresponding vector, v. Associated with each space compaction column is a unique key element which is used to construct all member elements of the column. Tables 3.1 and 3.2 show the left and right response vector configurations for $(V = 3, n = 4, m = 4)$.

Definition 3.4 A *space compaction column* is the set of data bits whose modulo-2 sum forms one bit of a space compaction sequence.

Tables 3.1 and 3.2 show a three vector configuration. A specific feature of note is the *overlapping* compaction columns between individual response vectors. Specifically, every two consecutive response vectors share $(m - 1)$ compaction columns. A compaction column can have elements from two consecutive vectors.

										$E^3_{4,1}$	$E^3_{4,2}$	$E^3_{4,3}$	$\mathbf{E^3_{4,4}}$		
									$E^3_{3,1}$	$E^3_{3,2}$	$E^3_{3,3}$	$\mathbf{E^3_{3,4}}$			
								$E^3_{2,1}$	$E^3_{2,2}$	$E^3_{2,3}$	$\mathbf{E^3_{2,4}}$				
							$E^3_{1,1}$	$E^3_{1,2}$	$E^3_{1,3}$	$\mathbf{E^3_{1,4}}$					
						$E^2_{4,1}$	$E^2_{4,2}$	$E^2_{4,3}$	$\mathbf{E^2_{4,4}}$						
					$E^2_{3,1}$	$E^2_{3,2}$	$E^2_{3,3}$	$\mathbf{E^2_{3,4}}$							
				$E^2_{2,1}$	$E^2_{2,2}$	$E^2_{2,3}$	$\mathbf{E^2_{2,4}}$								
			$E^2_{1,1}$	$E^2_{1,2}$	$E^2_{1,3}$	$\mathbf{E^2_{1,4}}$									
			$E^1_{4,1}$	$E^1_{4,2}$	$E^1_{4,3}$	$\mathbf{E^1_{4,4}}$									
		$E^1_{3,1}$	$E^1_{3,2}$	$E^1_{3,3}$	$\mathbf{E^1_{3,4}}$										
	$E^1_{2,1}$	$E^1_{2,2}$	$E^1_{2,3}$	$\mathbf{E^1_{2,4}}$											
$\mathbf{E^1_{1,1}}$	$\mathbf{E^1_{1,2}}$	$\mathbf{E^1_{1,3}}$	$\mathbf{E^1_{1,4}}$												
L^1_1	L^1_2	L^1_3	L^1_4	L^1_5	L^1_6	L^1_7									
				L^2_1	L^2_2	L^2_3	L^2_4	L^2_5	L^2_6	L^2_7					
								L^3_1	L^3_2	L^3_3	L^3_4	L^3_5	L^3_6	L^3_7	

Table 3.1: Left shifting MISR space compaction

Example 3.2 In table 3.1, compaction columns L^2_1, L^2_2, L^2_3 overlap columns L^1_5, L^1_6, L^1_7, respectively; likewise, columns L^3_1, L^3_2, L^3_3 overlap columns L^2_5, L^2_6, L^2_7.

Definition 3.5 The *key elements* of the left space compaction columns are $E^1_{1,1}$, $E^1_{1,2}$, ..., $E^1_{1,m}$, and $E^v_{i,m}$, where $i = 1; 2, 3, \ldots, n$ and $v = 1, 2, 3, \ldots, V$.

Definition 3.6 The *key elements* of the right space compaction are $E^1_{1,1}$, $E^1_{1,2}$, ..., $E^1_{1,m}$, and $E^v_{i,1}$, where $i = 1, 2, 3, \ldots, n$ and $v = 1, 2, 3, \ldots, V$.

Example 3.3 The key elements of the response vectors in tables 3.1 and 3.2 are shown in bold. For instance, the key element of column L^2_7 is $E^2_{4,4}$ (table 3.1), and that of R^2_4 is $E^2_{1,1}$ (table 3.2).

Definition 3.7 Given a key element, $E^v_{i,j}$, the *column members* are the elements in the corresponding space compaction column.

Example 3.4 For key element $E^2_{1,1}$ in table 3.2, the members of the space compaction column in the right compaction are: { $E^2_{1,1}$, $E^2_{2,2}$, $E^2_{3,3}$, $E^2_{4,4}$ }.

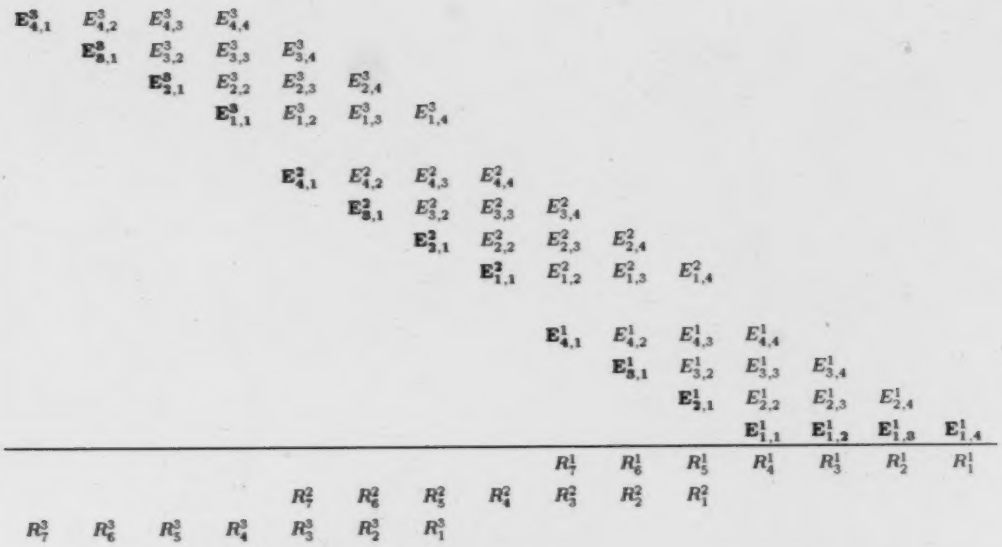

Table 3.2: Right shifting MISR space compaction

Definition 3.8 The *left* and *right space compaction sequence error masks*, denoted as $A_L(x)$ and $A_R(x)$ are defined as:

$$A_L(x) = M_L(x) \oplus M_L^*(x)$$

$$A_R(x) = M_R(x) \oplus M_R^*(x).$$

$A_L(x)$ and $A_R(x)$ are the left and right space compaction sequence error mask for the entire test length V; the subsequences of $A_L(x)$ and $A_R(x)$, denoted respectively as $A_L^v(x)$ and $A_R^v(x)$ $(1 \leq v \leq V)$, are the left and right space compaction sequence of vector v.

An individual subsequences $A_L^v(x)$ is extracted from $A_L(x)$ by right shifting $A_L(x)$ $((v-1) \times m)$ times and considering the least significant $(n+m-1)$ bits. Likewise, $A_R^v(x)$ can be extracted from $A_R(x)$.

Having extracted the subsequences, there is a direct relation between bits in a space compaction subsequences and the value of space compaction columns, as follows:

$$L_i^v = A_L^v[i],$$

$$R_i^v = A_R^v[i].$$

Example 3.5 Consider tables 3.1 and 3.2. They depict a scenario with three (4×4) response vectors. If the left and right space compaction sequence error masks are, respectively, $A_L(x) = 000111001000101$ and $A_R(x) = 001100100000010$, the corresponding subsequences are:

$$A_L^1(x) = 1000101,\ A_R^1(x) = 0000010$$

$$A_L^2(x) = 1100100,\ A_R^2(x) = 0010000$$

$$A_L^3(x) = 0001110,\ A_R^3(x) = 0011001.$$

The following definitions pertain specifically to entities used by the error identification algorithms.

Definition 3.9 The *Boolean equation set* is the set of equations formulated from the left and right space compaction columns of a response vector. The equations in the set can be solved simultaneous to establish the values of elements in the vector.

Definition 3.10 The *solution set* is a set of equations with each equation consisting of an element equal to a bit value. This set is the result of simultaneously solving a set of Boolean equations.

Definition 3.11 An *inconsistent* set of equations is a set with two or more equations that imply contradictory values for one or more variables.

Example 3.6 The set $B = \{\ E_{1,1}^1 = 1,\ E_{1,1}^1 \oplus E_{4,4}^1 = 0,\ E_{4,4}^1 = 0\ \}$ is inconsistent since both $E_{1,1}^1 = 1$ and $E_{1,1}^1 = 0$ are implied.

3.3 Two-known-stream Error Identification Algorithm

The proposed algorithm is able to identify all erroneous bits in a $(n \times m)$ response vector under the following constraints: (a) the m input data streams to the MISRs are independent, i.e. the errors in one stream do not affect the others, (b) errors are

localized in at most two data streams, and (c) the data streams containing errors are known.

As previously defined, a compaction column is a set of data bits whose modulo-2 sum forms one bit of a space compaction sequence. Thus each column can be written as a Boolean sum of up to m elements (the unknowns) equal to the respective bit in a space compaction sequence. For example, in table 3.2, assuming compaction column R_6^1 corresponds to a bit value of 1 in a space compaction sequence, the resulting equation would be: $(E_{3,1}^1 \oplus E_{4,2}^1 \oplus E_{1,3}^2 \oplus E_{2,4}^2 = 1)$. Given a set of such equations, they could be solved simultaneously to obtain the value of each element.

The total number of elements in a response vector, and hence the number of unknowns, is $(n \times m)$. The number of possible equations per $(n \times m)$ response vector is the number of compaction columns in the left and right space compaction, i.e. $(2(n+m-1))$. This system of equations cannot uniquely determine $(n \times m)$ unknowns. Thus the larger problem of uniquely determining the entire $(n \times m)$ vector is impossible given only the information contained in the two space compaction sequences. Instead we reduce the problem and require that all erroneous data bits occur in at most *two known* data streams (assumptions (b) and (c) above).

By limiting the focus to at most two erroneous data streams, the number of unknown elements is at most $(2 \times n)$. The values of the other $((m-2) \times n)$ elements can be computed by logic simulation of the good circuit, i.e. the error-free values. The number of equations per vector remains the same at $(2(n+m-1))$. Next, consider the error masks of the data bits, instead of the actual values, as follows:

$$E_{error-mask} = E_{error-free} \oplus E_{erroneous}.$$

The $((m-2) \times n)$ non-erroneous elements then attain a value of 0 and can be dropped from the equations. Likewise, the space compaction sequence error masks must be used in the equations instead of the space compaction sequences. Thus we have a system of $(2(n+m-1))$ equations in $(2 \times n)$ unknowns. This system can be solved simultaneously to uniquely determine the error mask value of each unknown element. The actual value of each element can then be computed as:

$$E_{actual-value} = E_{error-mask} \oplus E_{error-free}.$$

To summarize, for the case ($V = 1$), error bits in a ($n \times m$) vector are to be identified. The vector can be viewed as a left and right space compaction, respectively divided into left and right space compaction columns.

Each space compaction column has a value provided by a bit in the respective left or right space compaction error mask. The following points illustrate the error identification method:

1. There are at most ($2 \times n$) elements in the erroneous data streams.
2. The left and right compaction columns form a set of ($2(n + m - 1)$) Boolean equations in ($2 \times n$) unknowns (the elements in the erroneous data streams).
3. The value of each equation is the corresponding bit of the left or right space compaction sequence error mask.
4. The set of equations can be solved simultaneously to obtain unique error mask values for each of the ($2 \times n$) elements.
5. The actual value of each element is: $E_{actual-value} = E_{error-mask} \oplus E_{error-free}$.

The above method holds for one vector in isolation, i.e. ($V = 1$). However, for ($V > 1$), a compaction column can have member elements from two adjacent vectors, see tables 3.1 and 3.2. The number of equations for each vector is always ($2(n + m - 1)$). However, the number of elements, and thus unknowns, is no longer ($2 \times n$).

For response vector v where ($v = 1$ or $v = V$), there is one adjacent vector. The number of elements from vector v is ($2 \times n$). Additionally, the left compaction can contribute at most ($m - 1$) elements from the adjacent vector and likewise the right compaction, for an extra ($2(m - 1)$) elements. Thus, the total number of elements, and hence unknowns, is ($2 \times n + 2(m - 1) = 2(n + m - 1)$). These can be uniquely determined by ($2(n + m - 1)$) equations.

For each vector v where ($1 < v < V$), there are two adjacent response vectors, ($v-1$) and ($v+1$). The two adjacent vectors can contribute a maximum of ($2(m-1)$) elements per space compaction for a total of

$$2 \times n + 2(2(m - 1)) = 2(n + 2(m - 1))$$

elements. These cannot be uniquely determined by only $(2(n+m-1))$ equations. However, if vector $(v-1)$ was previously solved, the values of all elements in this vector are established. The solution equations for elements in vector $(v-1)$ that appear in the equations for vector v, numbering at most $(2(m-1))$, can be added to the set of $(2(n+m-1))$ equations for a total of $(2(n+2(m-1)))$ equations. This enhanced set can uniquely determine the $(2(n+2(m-1)))$ elements. Since vector $(v=1)$ can be successfully solved on its own, this method has a starting point, and all subsequent vectors can be solved in succession.

Error identification proceeds one response vector at a time. Although the complete set of equations for multiple response vectors can be written and solved simultaneously, this becomes infeasible with large test lengths consisting of several thousands of test vectors. Instead, each vector is solved individually. The left and right space compaction sequence error masks for the entire test length are respectively, A_L and A_R. To perform error identification, individual subsequences, A_L^v and A_R^v, for vector v must be extracted from A_L and A_R, respectively. A_L^v is simply the $(n+m-1)$ least significant bits of A_L right-shifted $((v-1)\times n)$ times; A_R^v is likewise obtained from A_R.

3.3.1 Algorithm

In the following algorithm, c_1 and c_2 are the erroneous data streams with $(1 \le c_1 < c_2 \le m)$, B is the set of Boolean equations, and $Solution^{v-1}$ and $Solution^v$ are the solution sets for response vectors $(v-1)$ and v, respectively. The subfunctions, *LKE()*, *RKE()*, *LCM()* and *RCM()*, compute the left and right key elements and column membership, respectively. They are defined following algorithm *Identify2Known()*.

Function *Identify2Known*(A_L^v, A_R^v, c_1, c_2, $Solution^{v-1}$)
/* computes the values of elements in data streams c_1 and c_2 of vector v */
/* with compaction sequence error masks A_L^v and A_R^v */
/* $Solution^{v-1}$ is the solution set for the previous vector */
    $B \leftarrow$ extract equations from $Solution^{v-1}$
    for $k = 1$ **to** $(n+m-1)$ **do**

$KeyElement_L \leftarrow LKE(L_k^v)$

$Column_L \leftarrow LCM(KeyElement_L)$

$B \leftarrow B \cup Equation(Column_L, c_1, c_2, A_L^v[k])$

$KeyElement_R \leftarrow RKE(R_k^v)$

$Column_R \leftarrow RCM(KeyElement_R)$

$B \leftarrow B \cup Equation(Column_R, c_1, c_2, A_R^v[k])$

$Solution^v \leftarrow$ solve B

/* an inconsistent set of equations is indicated via a flag */

return $Solution^v$ or inconsistent flag

end . /* *Identify2Known*() */

Function $LKE(L_k^v)$

/* computes the key element of the left compaction column L_k^v */

if $(v == 1)$ **then**

if $(k \leq m)$ **then**

return $E_{1,k}^v$

else

return $E_{k-m+1,m}^v$

else

if $(k < m)$ **then**

return $LKE(L_{n+k}^{v-1})$

else

return $E_{k-m+1,m}^v$

end . /* *LKE*() */

Function $RKE(R_k^v)$

/* computes the key element of the right compaction column R_k^v */

if $(v == 1)$ **then**

if $(k \leq m)$ **then**

```
            return E^v_{1,m-k+1}
        else
            return E^v_{k-m+1,1}
    else
        if (k < m) then
            return RKE(R^{v-1}_{n+k})
        else
            return E^v_{k-m+1,1}
end . /* RKE() */
```

The quantity, $LastVector$, in the following two functions, is the number of the last response vector to be recovered. For example, in a failing block of 256 response vectors, $LastVector = 256$.

```
Function LCM(E^v_{i,j})
/* computes the set of column members of the left compaction */
/* column given by key element E^v_{i,j} */
    s ← i; t ← j; C ← ∅
    if (v == LastVector) then
        while ((s ≤ n) AND (t ≥ 1))
            C ← C ∪ E^v_{s,t}
            s ← s + 1
            t ← t − 1
    else
        while (t ≥ 1)
            if (s ≤ n) then
                C ← C ∪ E^v_{s,t}
            else
                β ← s mod n
                C ← C ∪ E^{v+1}_{β,t}
```

$s \leftarrow \quad + 1$

$t \leftarrow i - 1$

/* C is the set of column elements */

return C

end . /* *LCM*() */

Function $RCM(E^v_{i,j})$

/* computes the set of column members of the right compaction */

/* column given by key element $E^v_{i,j}$ */

$s \leftarrow i$; $t \leftarrow j$; $C \leftarrow \emptyset$

if (v == *LastVector*) **then**

while (($s \leq n$) AND ($t \leq m$))

$C \leftarrow C \cup E^v_{s,t}$

$s \leftarrow s + 1$

$t \leftarrow t + 1$

else

while ($t \leq m$)

if ($s \leq n$) **then**

$C \leftarrow C \cup E^v_{s,t}$

else

$\beta \leftarrow s$ **mod** n

$C \leftarrow C \cup E^{v+1}_{\beta,t}$

$s \leftarrow s + 1$

$t \leftarrow t + 1$

/* C is the set of column elements */

return C

end . /* *RCM*() */

Function *Equation*(C, c_1, c_2, *value*)

/* formulates the Boolean equation for a compaction column */

/* C is a set of member elements of a compaction column */

/* c_1, c_2 are the erroneous data streams /*

/* *value* is a bit-value of a space compaction sequence */

/* the function returns the formulated Boolean equation */

if ($E^{v_1}_{a,c_1} \in C$ AND $E^{v_2}_{b,c_1} \in C$) **then**

return $\{E^{v_1}_{a,c_1} \oplus E^{v_2}_{b,c_2} = value\}$

else if ($E^{v_1}_{a,c_1} \in C$) **then**

return $\{E^{v_1}_{a,c_1} = value\}$

else if ($E^{v_2}_{b,c_2} \in C$) **then**

return $\{E^{v_2}_{b,c_2} = value\}$

else

return $\{0 = value\}$

end . /* *Equation*() */

3.3.2 Error Identification Example

Consider the space compactions depicted in tables 3.1 and 3.2 ($V = 3, n = 4, m = 4$) with the following data bits in error: $E^1_{1,1}$, $E^1_{3,1}$, $E^1_{2,4}$, $E^1_{4,4}$, $E^2_{1,1}$, $E^2_{2,4}$, $E^2_{4,4}$, $E^3_{1,1}$, $E^3_{2,1}$ and $E^3_{1,4}$. These errors correspond to the compaction sequence error masks, $A_L = 000111001000101$ and $A_R = 001100100000010$, and the erroneous data streams, $c_1 = 1$ and $c_2 = 4$.

Since $V = 3$, three applications of algorithm *Identify2Known()* are required to perform the complete error identification.

Vector 1: ($v = 1$, $A^1_L = 1000101$, $A^1_R = 0000010$, $Solution^0 = \emptyset$)

Since ($Solution^0 = \emptyset$), there are no equations from a previous solution set to include in B; therefore, initially $B = \emptyset$.

Table 3.3 details the Boolean equations added to B from the corresponding left and right space compaction columns. The complete set of Boolean equations is: $B =$ { $E^1_{1,1}$=1, $E^1_{2,1}$=0, $E^1_{3,1}$=1, $E^1_{4,1} \oplus E^1_{1,4}$=0, $E^2_{1,1} \oplus E^1_{2,4}$=0, $E^2_{2,1} \oplus E^1_{3,4}$=0, $E^2_{3,1} \oplus E^1_{4,4}$=1,

Column	Key Element	Members	Equation	Value
L_1^1	$E_{1,1}^1$	$E_{1,1}^1$	$E_{1,1}^1 =$	1
L_2^1	$E_{1,2}^1$	$E_{2,1}^1\ E_{1,2}^1$	$E_{2,1}^1 =$	0
L_3^1	$E_{1,3}^1$	$E_{3,1}^1\ E_{2,2}^1\ E_{1,3}^1$	$E_{3,1}^1 =$	1
L_4^1	$E_{1,4}^1$	$E_{4,1}^1\ E_{3,2}^1\ E_{2,3}^1\ E_{1,4}^1$	$E_{4,1}^1 \oplus E_{1,4}^1 =$	0
L_5^1	$E_{2,4}^1$	$E_{1,1}^2\ E_{4,2}^1\ E_{3,3}^1\ E_{2,4}^1$	$E_{1,1}^2 \oplus E_{2,4}^1 =$	0
L_6^1	$E_{3,4}^1$	$E_{2,1}^2\ E_{1,2}^2\ E_{4,3}^1\ E_{3,4}^1$	$E_{2,1}^2 \oplus E_{3,4}^1 =$	0
L_7^1	$E_{4,4}^1$	$E_{3,1}^2\ E_{2,2}^2\ E_{1,3}^2\ E_{4,4}^1$	$E_{3,1}^2 \oplus E_{4,4}^1 =$	1
R_1^1	$E_{1,4}^1$	$E_{1,4}^1$	$E_{1,4}^1 =$	0
R_2^1	$E_{1,3}^1$	$E_{1,3}^1\ E_{2,4}^1$	$E_{2,4}^1 =$	1
R_3^1	$E_{1,2}^1$	$E_{1,2}^1\ E_{2,3}^1\ E_{3,4}^1$	$E_{3,4}^1 =$	0
R_4^1	$E_{1,1}^1$	$E_{1,1}^1\ E_{2,2}^1\ E_{3,3}^1\ E_{4,4}^1$	$E_{1,1}^1 \oplus E_{4,4}^1 =$	0
R_5^1	$E_{2,1}^1$	$E_{2,1}^1\ E_{3,2}^1\ E_{4,3}^1\ E_{1,4}^2$	$E_{2,1}^1 \oplus E_{1,4}^2 =$	0
R_6^1	$E_{3,1}^1$	$E_{3,1}^1\ E_{4,2}^1\ E_{1,3}^2\ E_{2,4}^2$	$E_{3,1}^1 \oplus E_{2,4}^2 =$	0
R_7^1	$E_{4,1}^1$	$E_{4,1}^1\ E_{1,2}^2\ E_{2,3}^2\ E_{3,4}^2$	$E_{4,1}^1 \oplus E_{3,4}^2 =$	0

Table 3.3: Equations derived for vector ($v = 1$)

$E_{1,4}^1$=0, $E_{2,4}^1$=1, $E_{3,4}^1$=0, $E_{1,1}^1 \oplus E_{4,4}^1$=0, $E_{2,1}^1 \oplus E_{1,4}^2$=0, $E_{3,1}^1 \oplus E_{2,4}^2$=0, $E_{4,1}^1 \oplus E_{3,4}^2$=0 }.

Solving the set of equations B results in $Solution^1$ =

{ $E_{1,1}^1$=1, $E_{2,1}^1$=0, $E_{3,1}^1$=1, $E_{4,1}^1$=0, $E_{1,1}^2$=1, $E_{2,1}^2$=0, $E_{3,1}^2$=0, $E_{1,4}^1$=0, $E_{2,4}^1$=1, $E_{3,4}^1$=0, $E_{4,4}^1$=1, $E_{1,4}^2$=0, $E_{2,4}^2$=1, $E_{3,4}^2$=0 }.

Vector 2: ($v = 2$, $A_L^2 = 1100100$, $A_R^2 = 0010000$)

Initially, the solution equations for elements from vector 1 that appear in equations for vector 2 are added to B, i.e.:

B = { $E_{2,1}^1$=0, $E_{3,1}^1$=1, $E_{4,1}^1$=0, $E_{2,4}^1$=1, $E_{3,4}^1$=0, $E_{4,4}^1$=1 }.

Table 3.4 details the Boolean equations added to B from the corresponding left and right compaction columns. The complete set of Boolean equations is B =

{ $E_{2,1}^1$=0, $E_{3,1}^1$=1, $E_{4,1}^1$=0, $E_{2,4}^1$=1, $E_{3,4}^1$=0, $E_{4,4}^1$=1, $E_{1,1}^2 \oplus E_{2,4}^1$=0, $E_{2,1}^2 \oplus E_{3,4}^1$=0, $E_{3,1}^2 \oplus E_{4,4}^1$=1, $E_{4,1}^2 \oplus E_{1,4}^2$=0, $E_{1,1}^3 \oplus E_{2,4}^2$=0, $E_{2,1}^3 \oplus E_{3,4}^2$=1, $E_{3,1}^3 \oplus E_{4,4}^2$=1, $E_{2,1}^1 \oplus E_{1,4}^2$=0, $E_{3,1}^1 \oplus E_{2,4}^2$=0, $E_{4,1}^1 \oplus E_{3,4}^2$=0, $E_{1,1}^2 \oplus E_{4,4}^2$=0, $E_{2,1}^2 \oplus E_{1,4}^3$=1, $E_{3,1}^2 \oplus E_{2,4}^3$=0, $E_{4,1}^2 \oplus E_{3,4}^3$=0 }.

Solving the set of equations B results in $Solution^2$ =

{ $E_{2,1}^1$=0, $E_{3,1}^1$=1, $E_{4,1}^1$=0, $E_{1,1}^2$=1, $E_{2,1}^2$=0, $E_{3,1}^2$=0, $E_{4,1}^2$=0, $E_{1,1}^3$=1, $E_{2,1}^3$=1, $E_{3,1}^3$=0, $E_{2,4}^1$=1, $E_{3,4}^1$=0, $E_{4,4}^1$=1, $E_{1,4}^2$=0, $E_{2,4}^2$=1, $E_{3,4}^2$=0, $E_{4,4}^2$=1, $E_{1,4}^3$=1, $E_{2,4}^3$=0, $E_{3,4}^3$=0 }.

Vector 3: ($v = 3$, $A_L^3 = 0001110$, $A_R^3 = 0011001$)

Initially, the solution equations for elements from vector 2 that appear in equations

Column	Key Element	Members	Equation	Value
L_1^2	$E_{2,4}^1$	$E_{1,1}^2\ E_{4,2}^1\ E_{3,3}^1\ E_{2,4}^1$	$E_{1,1}^2 \oplus E_{2,4}^1 =$	0
L_2^2	$E_{3,4}^1$	$E_{2,1}^2\ E_{1,2}^2\ E_{4,3}^1\ E_{3,4}^1$	$E_{2,1}^2 \oplus E_{3,4}^1 =$	0
L_3^2	$E_{4,4}^1$	$E_{3,1}^2\ E_{2,2}^2\ E_{1,3}^1\ E_{4,4}^1$	$E_{3,1}^2 \oplus E_{4,4}^1 =$	1
L_4^2	$E_{1,4}^2$	$E_{4,1}^2\ E_{3,2}^2\ E_{2,3}^2\ E_{1,4}^2$	$E_{4,1}^2 \oplus E_{1,4}^2 =$	0
L_5^2	$E_{2,4}^2$	$E_{1,1}^3\ E_{4,2}^2\ E_{3,3}^2\ E_{2,4}^2$	$E_{1,1}^3 \oplus E_{2,4}^2 =$	0
L_6^2	$E_{3,4}^2$	$E_{2,1}^3\ E_{1,2}^3\ E_{4,3}^2\ E_{3,4}^2$	$E_{2,1}^3 \oplus E_{3,4}^2 =$	1
L_7^2	$E_{4,4}^2$	$E_{3,1}^3\ E_{2,2}^3\ E_{1,3}^3\ E_{4,4}^2$	$E_{3,1}^3 \oplus E_{4,4}^2 =$	1
R_1^2	$E_{2,1}^1$	$E_{2,1}^1\ E_{3,2}^1\ E_{4,3}^1\ E_{1,4}^2$	$E_{2,1}^1 \oplus E_{1,4}^2 =$	0
R_2^2	$E_{3,1}^1$	$E_{3,1}^1\ E_{4,2}^1\ E_{1,3}^2\ E_{2,4}^2$	$E_{3,1}^1 \oplus E_{2,4}^2 =$	0
R_3^2	$E_{4,1}^1$	$E_{4,1}^1\ E_{1,2}^2\ E_{2,3}^2\ E_{3,4}^2$	$E_{4,1}^1 \oplus E_{3,4}^2 =$	0
R_4^2	$E_{1,1}^2$	$E_{1,1}^2\ E_{2,2}^2\ E_{3,3}^2\ E_{4,4}^2$	$E_{1,1}^2 \oplus E_{4,4}^2 =$	0
R_5^2	$E_{2,1}^2$	$E_{2,1}^2\ E_{3,2}^2\ E_{4,3}^2\ E_{1,4}^3$	$E_{2,1}^2 \oplus E_{1,4}^3 =$	1
R_6^2	$E_{3,1}^2$	$E_{3,1}^2\ E_{4,2}^2\ E_{1,3}^3\ E_{2,4}^3$	$E_{3,1}^2 \oplus E_{2,4}^3 =$	0
R_7^2	$E_{4,1}^2$	$E_{4,1}^2\ E_{1,2}^3\ E_{2,3}^3\ E_{3,4}^3$	$E_{4,1}^2 \oplus E_{3,4}^3 =$	0

Table 3.4: Equations derived for vector ($v = 2$)

for vector 3 are added to B, i.e.:

$B = \{\ E_{2,1}^2=0,\ E_{3,1}^2=0,\ E_{4,1}^2=0,\ E_{2,4}^2=1,\ E_{3,4}^2=0,\ E_{4,4}^2=1\ \}$.

Table 3.5 details the Boolean equations added to B from the corresponding left and right compaction columns. The complete set of Boolean equations is $B =$

$\{\ E_{2,1}^2=0,\ E_{3,1}^2=0,\ E_{4,1}^2=0,\ E_{2,4}^2=1,\ E_{3,4}^2=0,\ E_{4,4}^2=1,\ E_{1,1}^3 \oplus E_{2,4}^2=0,\ E_{2,1}^3 \oplus E_{3,4}^2=1,$
$E_{3,1}^3 \oplus E_{4,4}^2=1,\ E_{4,1}^3 \oplus E_{1,4}^3=1,\ E_{2,4}^3=0,\ E_{3,4}^3=0,\ E_{4,4}^3=0,\ E_{2,1}^2 \oplus E_{1,4}^3=1,\ E_{3,1}^2 \oplus E_{2,4}^3=0,$
$E_{4,1}^2 \oplus E_{3,4}^3=0,\ E_{1,1}^3 \oplus E_{4,4}^3=1,\ E_{2,1}^3=1,\ E_{3,1}^3=0,\ E_{4,1}^3=0\ \}$.

Solving the set of equations B results in $Solution^3 =$

$\{\ E_{2,1}^2=0,\ E_{3,1}^2=0,\ E_{4,1}^2=0,\ E_{1,1}^3=1,\ E_{2,1}^3=1,\ E_{3,1}^3=0,\ E_{4,1}^3=0,\ E_{2,4}^2=1,\ E_{3,4}^2=0,\ E_{4,4}^2=1,$
$E_{1,4}^3=1,\ E_{2,4}^3=0,\ E_{3,4}^3=0,\ E_{4,4}^3=0\ \}$.

The elements equal to 1 within the sets $Solution^1$, $Solution^2$ and $Solution^3$ are the erroneous data bits: $E_{1,1}^1$, $E_{3,1}^1$, $E_{2,4}^1$, $E_{4,4}^1$, $E_{1,1}^2$, $E_{2,4}^2$, $E_{4,4}^2$, $E_{1,1}^3$, $E_{2,1}^3$ and $E_{1,4}^3$.

3.3.3 Complexity Analysis

The computational complexity of an algorithm [4] is proportional to the total number of basic operations performed for a specific input size, n. Computational complexity shows how the execution time grows as a function of the input size. In addition to

Column	Key Element	Members	Equation	Value
L_1^3	$E_{2,4}^2$	$E_{1,1}^3\ E_{4,2}^2\ E_{3,3}^2\ E_{2,4}^2$	$E_{1,1}^3 \oplus E_{2,4}^2 =$	0
L_2^3	$E_{3,4}^2$	$E_{2,1}^3\ E_{1,2}^3\ E_{4,3}^2\ E_{3,4}^2$	$E_{2,1}^3 \oplus E_{3,4}^2 =$	1
L_3^3	$E_{4,4}^2$	$E_{3,1}^3\ E_{2,2}^3\ E_{1,3}^3\ E_{4,4}^2$	$E_{3,1}^3 \oplus E_{4,4}^2 =$	1
L_4^3	$E_{1,4}^3$	$E_{4,1}^3\ E_{3,2}^3\ E_{2,3}^3\ E_{1,4}^3$	$E_{4,1}^3 \oplus E_{1,4}^3 =$	1
L_5^3	$E_{2,4}^3$	$E_{4,2}^3\ E_{3,3}^3\ E_{2,4}^3$	$E_{2,4}^3 =$	0
L_6^3	$E_{3,4}^3$	$E_{4,3}^3\ E_{3,4}^3$	$E_{3,4}^3 =$	0
L_7^3	$E_{4,4}^3$	$E_{4,4}^3$	$E_{4,4}^3 =$	0
R_1^3	$E_{2,1}^2$	$E_{2,1}^2\ E_{3,2}^2\ E_{4,3}^2\ E_{1,4}^3$	$E_{2,1}^2 \oplus E_{1,4}^3 =$	1
R_2^3	$E_{3,1}^2$	$E_{3,1}^2\ E_{4,2}^2\ E_{1,3}^3\ E_{2,4}^3$	$E_{3,1}^2 \oplus E_{2,4}^3 =$	0
R_3^3	$E_{4,1}^2$	$E_{4,1}^2\ E_{1,2}^3\ E_{2,3}^3\ E_{3,4}^3$	$E_{4,1}^2 \oplus E_{3,4}^3 =$	0
R_4^3	$E_{1,1}^3$	$E_{1,1}^3\ E_{2,2}^3\ E_{3,3}^3\ E_{4,4}^3$	$E_{1,1}^3 \oplus E_{4,4}^3 =$	1
R_5^3	$E_{2,1}^3$	$E_{2,1}^3\ E_{3,2}^3\ E_{4,3}^3$	$E_{2,1}^3 =$	1
R_6^3	$E_{3,1}^3$	$E_{3,1}^3\ E_{4,3}^3$	$E_{3,1}^3 =$	0
R_7^3	$E_{4,1}^3$	$E_{4,1}^3$	$E_{4,1}^3 =$	0

Table 3.5: Equations derived for vector ($v = 3$)

the input size, the execution time of an algorithm is also dependent on the input data set itself. Two equal length data sets may have different execution times. To remove this data dependency, the worst case scenario is assumed. For an input size, n, the maximum number of operations is considered.

Computational complexity is stated in big-O notation. This gives the asymptotic growth rate of the algorithm as a function of input size. For example, an $O(n^2)$ algorithm grows in complexity as the square of the input size.

Algorithm 3.3.1 is divided into two stages. The computational complexity of each stage is first given, then the complexity of the entire algorithm is stated as the sum of the complexities of the two stages.

The first stage is creating the set of Boolean equations. There are exactly $(2(n + m - 1))$ Boolean equations for any response vector. The number of elementary operations in creating an individual equation, in the worst case, is bounded by the size of an individual compaction column m. Thus this task takes $(2m(n+m-1))$ operations. But since typically $n >> m$ ($n = 1024, m = 16$ or $m = 22$), the complexity of this stage is $O(n)$.

The second stage is solving the system of equations. A standard method for solving a system of linear equations is Gaussian elimination. The complexity of

Gaussian elimination is $O(n^3)$ [3], where n is the order of the system. The typical system consists of $(2(n+m-1))$ equations in $(2(n+m-1))$ unknowns. This can be represented by a $(2(n+m-1)+1)$ by $(2(n+m-1))$ augmented matrix. If we again make the observation that $n >> m$, in the limit the complexity of Gaussian elimination for a $(2n \times 2n)$ system is $O(n^3)$.

Thus the complexity of the entire algorithm is $O(n) + O(n^3) = O(n^3)$. In other words, the computation time of algorithm 3.3.1 is proportional to the cube of the data stream length.

For the recovery of a failing block, consisting of 256 response vectors, algorithm 3.3.1 is invoked 256 times. However, Gaussian elimination, with its high computational complexity, is not required to solve all the Boolean equation sets generated. It can be seen that aside from the first and last vectors, the intermediate response vectors produce similar equation sets, differing only in the element superscript v. Thus, this equation set can be solved once using Gaussian elimination, and subsequently, back-substitution can be used to obtain the other solutions. Back-substitution is $O(n)$, i.e. linear with the input size. Thus the complexity of the algorithm in the case of multiple vectors is a one-time cost of $O(n^3)$ and $O(n)$ from then on, i.e. linear.

3.4 Two-unknown-stream Error Identification Algorithm

The second proposed error identification algorithm relaxes the constraint of knowing which data streams contain errors. Without this knowledge, the problem becomes one of solving $(n \times m)$ unknowns with only $(2(n+m-1))$ equations. For a single response vector this may lead to multiple possible solutions.

The new constraints for the second error identification algorithm are: (a) the m input data streams to the MISRs are independent, i.e. the errors in one stream do not affect the others, and (b) errors are localized in at most two data streams. Due to constraint (b) only solutions involving at most two data streams need be considered. Thus all potential solutions can be grouped according to the pairs of data streams they involve. Further, by considering the solutions of multiple response vectors, pairs

of data streams that fail to produce a consistent solution for any one vector can be eliminated. Algorithm 3.3.1 is used to test each suspect pair of data streams if it produces a consistent solution for every response vector.

Due to overlap between response vectors, an inconsistent equation set can only occur in the first or last vector of the test set. Recall, from example 3.3.2, the equations derived for vector ($v = 2$). Two types of equations are present: (1) equations with single elements from vector ($v = 1$), and (2) equations with exactly two elements derived from the space compaction columns in vector ($v = 2$). If the equations in (1) are consistent the entire set must be consistent. Equations derived for the first and last vectors may contain two, one or zero elements. Such equations can be inherently inconsistent (for example the equation "0=1") or they may render the set inconsistent when combined with equations from the previous solution set.

To quickly reduce the number of solutions produced by the algorithm by eliminating suspect data streams, a further constraint is added: (c) the number of errors in a response vector must be strictly less than ($2n$). Two error-free compaction sequences can only be produced by an error-free response vector or one with ($2n$) errors with specific errors in adjacent vectors. Tables 3.6 and 3.7 illustrated such an example for the configuration ($V = 3, n = 4, m = 4$); shown are the error masks of the response vectors and compaction sequences, a "1" indicates an error and "0" otherwise. Note the configuration of errors in data streams one and three that produce the error-free compaction sequences for response vector ($v = 2$). The constraint on the number of errors per response vector eliminates this possibility and permits another test of solved solutions. A solution indicating erroneous elements based on error-free space compaction sequences invalidates all solutions corresponding to the given suspect pair of data streams.

Initially, all ${}_mC_2$ pairs of data streams are potentially erroneous. Algorithm 3.3.1 is used to test all suspect combinations of data streams with each pair of compaction sequence error masks in succession. Algorithm 3.3.1 has been modified to return the solution of the system of equations or a flag indicating an inconsistent system. Should a solution be returned, it is stored indexed by the respective pair of suspect

											0	0	0	0	
										1	0	0	0		
									1	0	1	0			
								1	0	1	0				
							1	0	1	0					
						1	0	1	0						
					1	0	1	0							
				1	0	1	0								
			1	0	1	0									
		1	0	1	0										
	1	0	0	0											
0	0	0	0												
0	1	1	1	0	0	0									
				0	0	0	0	0	0	0					
								0	0	0	1	0	0	0	

Table 3.6: Error cancellations in the left compaction stream

data streams. If an inconsistent flag is returned or errors are identified based on error-free compaction sequences, the suspect pair of data streams is removed from further considerations and all previous, related solutions are discarded. Thus, at the start, all pairs of suspect data streams must be tested, but as further response vectors are solved, more suspect pairs of data streams are eliminated from consideration.

This algorithm may provide multiple solutions which comply with all $(2 \times V)$ space compaction sequence error masks. Error cancellation occurs when the space compaction process maps many $(n \times m)$ vectors into a single pair of space compaction sequences. Without additional information, it is then impossible to distinguish the truly erroneous $(n \times m)$ vector among the multiple possibilities returned by the error identification algorithm. To further narrow down the possibilities, structural analysis and fault simulation must be attempted.

3.4.1 Algorithm

In the following algorithm, *SuspectList* is the set of valid combinations of two streams and $S^v_{c_1,c_2}$ is the solution set for response vector v with erroneous streams c_1 and c_2.

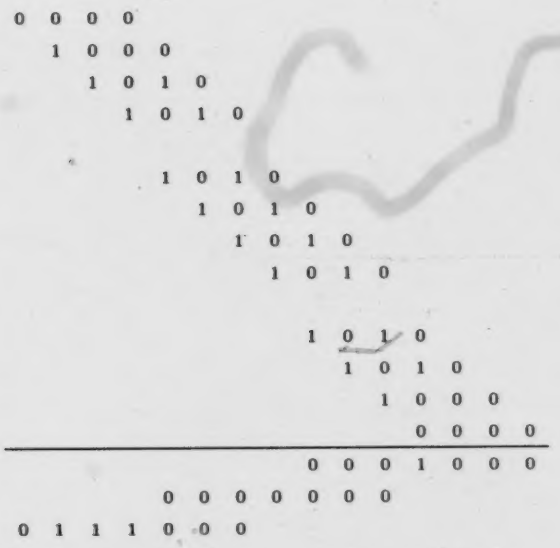

Table 3.7: Error cancellations in the right compaction stream

Function *Identify2Unknown*(A_L, A_R, $NumVectors$)

/* computes the values of elements determined by the space compaction */

/* sequence error masks A_L and A_R */

/* $NumVectors$ is the number of compacted vectors */

SuspectList $\leftarrow \emptyset$

for $c_1 = 1$ **to** $(m-1)$ **do**

for $c_2 = (c_1 + 1)$ **to** m **do**

$S^0_{c_1,c_2} \leftarrow \emptyset$

SuspectList $\leftarrow$ *SuspectList* $\cup \{c_1, c_2\}$

for $v = 1$ **to** $NumVectors$ **do**

$A^v_L \leftarrow$ *GetCompactionSequence*(A_L)

$A^v_R \leftarrow$ *GetCompactionSequence*(A_R)

for each $\{c_1, c_2\} \in$ *SuspectList* **do**

$S^v_{c_1,c_2} \leftarrow$ *Identify2Known*(A^v_L, A^v_R, c_1, c_2, $S^{v-1}_{c_1,c_2}$)

if (($S^v_{c_1,c_2}$ == inconsistent) OR

$(A_L^v == A_R^v == 0$ AND $ErrorCount(S_{c_1,c_2}^v) > 0))$

for $k = 0$ **to** v **do**

discard S_{c_1,c_2}^k

$SuspectList \leftarrow SuspectList - \{c_1, c_2\}$

return solutions for each valid combination of data streams

end . /* *Identify2Unknown*() */

3.4.2 Error Identification Example

Reconsider the example in 3.3.2. The space compactions depicted in tables 3.1 and 3.2 ($V = 3, n = 4, m = 4$) are again assumed with the following data bits in error: $E_{1,1}^1$, $E_{3,1}^1$, $E_{2,4}^1$, $E_{4,4}^1$, $E_{1,1}^2$, $E_{2,4}^2$, $E_{4,4}^2$, $E_{1,1}^3$, $E_{2,1}^3$ and $E_{1,4}^3$. These errors produce the space compaction sequence masks A_L = 000111001000101 and A_R = 001100100000010. However the erroneous data streams are unknown.

Algorithm 3.4.1 starts by initializing the list of suspect pairs of data streams: *SuspectList* = { {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4} }.

Vector 1: ($v = 1$, A_L^1 = 1000101, A_R^1 = 0000010)

Invoking *Identify2Known()* for each pair of data streams in *SuspectList* results in the following:

{1,2} produces an inconsistent set of equations,

{1,3} produces $Solution_{1,3}^1$ = { $E_{1,1}^1$=1, $E_{2,1}^1$=0, $E_{3,1}^1$=0, $E_{4,1}^1$=0, $E_{1,1}^2$=1, $E_{2,1}^2$=0, $E_{3,1}^2$=1, $E_{1,3}^1$=1, $E_{2,3}^1$=0, $E_{3,3}^1$=1, $E_{4,3}^1$=0, $E_{1,3}^2$=0, $E_{2,3}^2$=0 },

{1,4} produces $Solution_{1,4}^1$ = { $E_{1,1}^1$=1, $E_{2,1}^1$=0, $E_{3,1}^1$=1, $E_{4,1}^1$=0, $E_{1,1}^2$=1, $E_{2,1}^2$=0, $E_{3,1}^2$=0, $E_{1,4}^1$=0, $E_{2,4}^1$=1, $E_{3,4}^1$=0, $E_{4,4}^1$=1, $E_{1,4}^2$=0, $E_{2,4}^2$=1, $E_{3,4}^2$=0 },

{2,3} produces an inconsistent set of equations,

{2,4} produces an inconsistent set of equations,

{3,4} produces an inconsistent set of equations.

After error identification in vector 1, *SuspectList* is equal to { {1,3}, {1,4} }.

Vector 2: ($v = 2$, A_L^2 = 1100100, A_R^2 = 0010000)

Invoking *Identify2Known()* for each pair of streams in *SuspectList* results in the following potential solutions:

{1,3} produces $Solution^2_{1,3}$ = { $E^1_{2,1}$=0, $E^1_{3,1}$=0, $E^1_{4,1}$=0, $E^2_{1,1}$=1, $E^2_{2,1}$=0, $E^2_{3,1}$=1, $E^2_{4,1}$=0, $E^3_{1,1}$=1, $E^3_{2,1}$=0, $E^3_{3,1}$=0, $E^1_{3,3}$=1, $E^1_{4,3}$=0, $E^2_{1,3}$=0, $E^2_{2,3}$=0, $E^2_{3,3}$=1, $E^2_{4,3}$=1, $E^3_{1,3}$=1, $E^3_{2,3}$=0 },

{1,4} produces $Solution^2_{1,4}$ = { $E^1_{2,1}$=0, $E^1_{3,1}$=1, $E^1_{4,1}$=0, $E^2_{1,1}$=1, $E^2_{2,1}$=0, $E^2_{3,1}$=0, $E^2_{4,1}$=0, $E^3_{1,1}$=1, $E^3_{2,1}$=1, $E^3_{3,1}$=0, $E^1_{2,4}$=1, $E^1_{3,4}$=0, $E^1_{4,4}$=1, $E^2_{1,4}$=0, $E^2_{2,4}$=1, $E^2_{3,4}$=0, $E^2_{4,4}$=1, $E^3_{1,4}$=1, $E^3_{2,4}$=0, $E^3_{3,4}$=0 }.

After error identification in vector 2, *SuspectList* is equal to { {1,3}, {1,4} }.

Vector 3: ($v = 3$, A^3_L = 0001110, A^3_R = 0011001)

Invoking *Identify2Known()* for each pair of streams in *SuspectList* results in the following potential solutions:

{1,3} produces an inconsistent set of equations, $Solution^1_{1,3}$ and $Solution^2_{1,3}$ are discarded,

{1,4} produces $Solution^3_{1,4}$ = { $E^2_{2,1}$=0, $E^2_{3,1}$=0, $E^2_{4,1}$=0, $E^3_{1,1}$=1, $E^3_{2,1}$=1, $E^3_{3,1}$=0, $E^3_{4,1}$=0, $E^2_{2,4}$=1, $E^2_{3,4}$=0, $E^2_{4,4}$=1, $E^3_{1,4}$=1, $E^3_{2,4}$=0, $E^3_{3,4}$=0, $E^3_{4,4}$=0 }.

After error identification in vector 3, *SuspectList* is equal to { {1, 4} }.

The ultimate result is a single pair of suspect data streams, {1, 4}, which produces the consistent solution sets $Solution^1_{1,4}$, $Solution^2_{1,4}$, and $Solution^3_{1,4}$. The elements equal to 1 within these sets are the erroneous data bits: $E^1_{1,1}$, $E^1_{3,1}$, $E^1_{2,4}$, $E^1_{4,4}$, $E^2_{1,1}$, $E^2_{2,4}$, $E^2_{4,4}$, $E^3_{1,1}$, $E^3_{2,1}$ and $E^3_{1,4}$.

3.4.3 Complexity Analysis

Algorithm 3.4.1 invokes algorithm *Identify2Known()* a number of times proportional to the number of suspect pairs of data streams. In the worst case, for a failing block of 256 response vectors, if no data streams were eliminated from consideration this entails ($_mC_2 \times 256$) calls of algorithm *Identify2Known()*. In the case of ($m = 16$) data streams, $_{16}C_2 = 120$. In the limit, the worst case complexity of algorithm 3.4.1 is $120 \times 256 \times O(n^3) = O(n^3)$.

3.5 Alternative Data Recovery Schemes

The primary data recovery scheme presented in section 3.1 differs from the diagnosis scheme in [19, 20] in terms of the data retrieval method used. It is based on the same data compaction process with an additional MISR, and uses intermediate signature comparison to locate failing blocks of circuit response. This section explores two alternatives to the primary data recovery scheme. The first alternative improves the resolutions of the error identification algorithms presented in sections 3.3.1 and 3.4.1. The second alternative greatly simplifies the STUMPS testing and data retrieval procedures by utilizing the merits of our proposed data construction technique. It has the potential to replace the conventional STUMPS-based testing and diagnosis procedures in practical applications.

3.5.1 With Non-overlapping Response Vectors

The preceding error identification algorithms are designed to work in a STUMPS testing environment. The compaction process was unmodified from the standard architecture but for the addition of a second MISR. Recall the model of MISR-based compaction described in section 3.1.1. Because data bits from two successive response vectors may be compacted together, $(m-1)$ identical compaction columns occur in both vectors. Example 3.2 lists the overlapping columns in the three response vectors of table 3.1.

The proposed alternative error identification method employs non-overlapping response vectors. To completely separate response vectors requires the insertion of $(m-1)$ all-zero states between two successive vectors. It can be achieved by supplying the MISRs with all-zero inputs and clocking them $(m-1)$ times. Figure 3.8 shows the resulting non-overlapping left space compaction for the configuration $(V=2, n=4, m=4)$ and the equivalent model.

The proposed modification increases the amount of information available to solve the system of equations in each circuit response vector. The new left and right $(n+m-1)$-bit space compaction sequences solely represent the data bits in one response vector. Whereas in the overlapping case, $(n-m+1)$ bits of each compaction

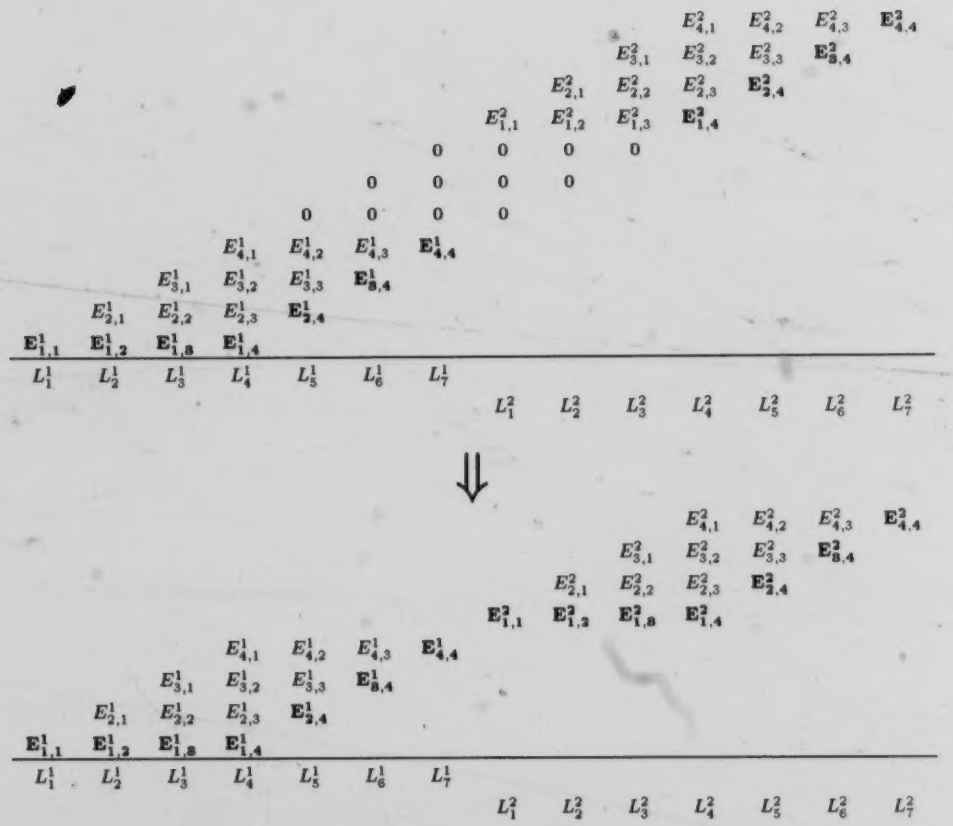

Table 3.8: Non-overlapping left shifting MISR space compaction

sequence determine a single vector, and $(2(m-1))$ bits are shared with adjacent vectors. As a result, the non-overlapping scheme significantly improves the error identification resolutions and reduces the computational time required for further fault diagnosis.

On the other hand, the non-overlapping scheme lengthens the test process by a factor of $(m-1)$. However, since $n >> m$, $(V(m-1))$ has only a slight impact on the test length. For example, for $m = 16$, $n = 1024$, and $V = 100,000$, the increase in the test length is only

$$\frac{V(m-1)}{V \times n} \cong 1.46\%,$$

which is minimal.

Experimental results comparing the standard and modified error identification methods are presented in chapter 4. Below are the modified algorithms that implement the non-overlapping error identification method. The modifications alter how key elements and column members (defined in section 3.2) are computed. The number of equations that must be solved simultaneous is unchanged. As the complexity of the original algorithms depends on the set of equations that must be solved, the complexity of the modified algorithms is thus unchanged.

Modified Algorithms

Function *NonoverlappingLKE*(L_k^v)

/* computes the key element of the left compaction column L_k^v */

    if ($k \leq m$) **then**

        return $E_{1,k}^v$

    else

        return $E_{k-m+1,m}^v$

end . /* *NonoverlappingLKE*() */

Function *NonoverlappingRKE*(R_k^v)

/* computes the key element of the right compaction column R_k^v */

    if ($k \leq m$) **then**

        return $E_{1,m-k+1}^v$

    else

        return $E_{k-m+1,1}^v$

end . /* *NonoverlappingRKE*() */

Function *NonoverlappingLCM*($E_{i,j}^v$)

/* computes the set of column members of the left compaction */

/* column given by key element $E_{i,j}^v$ */

    $s \leftarrow i; t \leftarrow j; C \leftarrow \emptyset$

```
    while ((s ≤ n) AND (t ≥ 1))
        C ← C ∪ E^v_{s,t}
            s ← s + 1
            t ← t − 1
    /* C is the set of column elements */
    return C
end . /* NonoverlappingLCM() */

Function NonoverlappingRCM(E^v_{i,j})
/* computes the set of column members of the right compaction */
/* column given by key element E^v_{i,j} */
    s ← i; t ← j; C ← ∅
    while ((s ≤ n) AND (t ≤ m))
        C ← C ∪ E^v_{s,t}
            s ← s + 1
            t ← t + 1
    /* C is the set of column elements */
    return C
end . /* NonoverlappingRCM() */
```

3.5.2 With Simplified Testing Procedure

The second alternative abandons intermediate signature testing and simply performs data recovery on the entire test set. The test set is applied without interruption while simultaneously capturing the quotient sequences, Q_L and Q_R, from the two MISRs. The sequences can be stored for further processing off-line or by using the system given on page 23, the space compaction sequence error masks, A_L and A_R, can be constructed in real-time during the test. Data recovery can then be performed on any portion of the compaction sequences that contain errors as revealed by A_L and A_R.

Recall the modelling of space compaction sequences in section 3.1.1. A space compaction sequence, $M(x)$, does not physically exist in MISR-based data compaction. Rather, $M(x)$ is constructed from the quotient sequence, $Q(x)$, obtained from the MISR during testing. The space compaction sequence has the property that individual bits of $M(x)$ reflect the values of local uncompacted circuit response data bits. It can thus be used to locate the erroneous responses in the test by comparison with the error-free space compaction sequence, $M^*(x)$. Recall that the space compaction sequence error mask, $A(x) = M(x) \oplus M^*(x)$, is used to locate erroneous bits in $M(x)$. $A(x)$ contains a "1" if the corresponding bit in $M(x)$ is erroneous and "0", otherwise. The quotient sequence, $Q(x)$, has no such error localization: a bit of $Q(x)$ is determined in part by all data bits compacted up to that point. After the compaction of an erroneous data bit, all subsequent bits of $Q(x)$ from the MISR will be perturbed, irrespective of upcoming error-free and erroneous data bits.

The advantages of this alternative scheme are simplified testing procedures and decreased tester usage. Eliminating intermediate signatures has a two-fold saving in storage and computation time. Tester memory requirements can be reduced along with the computation that generates the intermediate signatures. More significantly, because there is no comparison of intermediate signatures, testing can proceed uninterrupted from start to finish. The tester-CUT interactions that set and reset BIST resources during interval testing are eliminated (refer to the data retrieval process on page 17), simplifying testing and tester hardware complexity. Diagnosis information can be obtained during normal pass/fail testing, altogether eliminating a separate test to perform data retrieval. Thus, testing time can be significantly reduced while utilizing less complex testing equipment.

Chapter 4

Experimental Results

Two different experiments were conducted to evaluate the effectiveness of the data recovery schemes: data recovery of pseudorandomly generated faulty responses and data recovery in standard benchmark circuits. The first experiment demonstrates the feasibility of the data recovery scheme on large circuits responses, while the second experiment demonstrates resolving multiple solutions produced by the error identification algorithms.

In this chapter, section 4.1 presents the software system used to perform the simulation experiments. Section 4.2 describes and presents the first simulation. Section 4.3 describes and presents the second simulation. Lastly, section 4.4 details the implementation of the custom simulation software developed in this research.

4.1 System Overview

The environment used to carry out the simulation experiments consists of a custom software package named *DataRecovery* (DR) and the set of ISCAS85 [7] benchmark circuits. The software was developed using the GNU C++ complier, **g++**, under SunOS and Solaris. All simulations were executed on identical SUN Ultra 1 workstations each equipped with 128 Mbytes of RAM.

DR is a collection of software modules integrated to perform data recovery of simulated circuit responses (description of the specific software modules is given in section 4.4). It implements the primary data recovery scheme presented in section 3.1.

and the alternative recovery scheme described in section 3.5.1. The recovery schemes are tested by means of two simulations, described below, corresponding to the way in which faulty responses are generated.

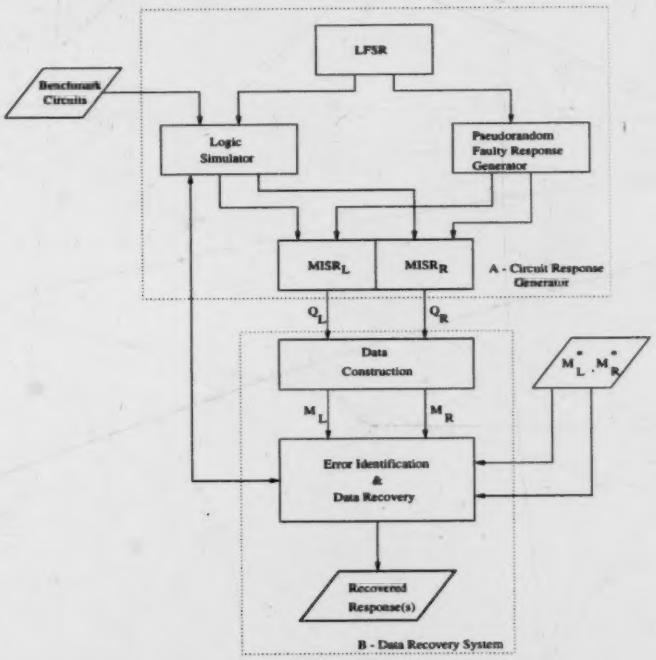

Figure 4.1: Software system block diagram

Figure 4.1 is a block diagram of the software system used to perform the two experiments. Shown are the significant system components and their interactions. The complete system can be logically divided into two subsystems: subsystem A generates and compacts circuit response vectors, and subsystem B performs data recovery on the partially compacted responses. The data exchanged between the subsystems is the quotient sequences, Q_L and Q_R. The two subsystems model a real testing environment consisting of the CUT and the tester, modeled by subsystem A, and the data recovery scheme, modeled by subsystem B.

The circuit response generator consists of the following components. The *LFSR* is used to generate pseudorandom sequences. It serves to create circuit responses and provide stimuli for the logic simulator. The *logic simulator* generates faulty responses by injecting stuck-at faults in the benchmark circuits. The generated responses are

then compacted by the $MISRs$. The *pseudorandom faulty response generator* uses the sequence of states from the LFSR to generate an arbitrary size, pseudorandom response vector. $MISR_L$ and $MISR_R$ compact the responses vectors from the simulator or pseudorandom faulty response generator to produce the quotient sequences, Q_L and Q_R, respectively.

The data recovery subsystem in turn consists of the following components. *Data construction* is the software implementation of the data construction technique. It computes the space compaction sequences, M_L and M_R, from the corresponding quotient sequences, Q_L and Q_R. *Error identification and data recovery* implement the error identification algorithms and combine the identified errors with the fault-free responses to produce the recovered responses. The *recovered responses* are the ultimate result of the data recovery scheme.

Two additional components exist in the complete system, outside the two defined subsystems. M_L^* and M_R^* are precomputed error-free space compaction sequences used during error identification. Lastly, the *benchmark circuits* are the ISCAS85 standard benchmarks that are used by the *logic simulator* to generate faulty responses.

The goal of the following experiments is to demonstrate the feasibility of the data recovery schemes presented in chapter 3. Feasibility is judged on the basis of computational time required by the error identification algorithms and the number of solutions recovered. The experiment assumptions are the constraints of error identification algorithm 3.4.1; i.e. at most two data streams may contain errors, errors are independent, and the number of errors per response vector is strictly less than $2n$, where n is the length of a data stream.

4.2 Data Recovery in Pseudorandomly Generated Response Vectors

The goal of the first simulation is to exercise the data recovery scheme on circuit responses of industrial sizes. Unfortunately, state-of-the-art VLSI designs are closely held company secrets and as such are unavailable for experimentation. Instead, data recovery is performed on response vectors created from the pseudorandom sequence

of states generated by an LFSR.

4.2.1 Simulation Environment

Referring to figure 4.1, the system components used in this simulation are as follows. Within subsystem A, response vectors are created by the *pseudorandom faulty response generator* from the sequence of states of the *LFSR*. The response vectors are then compacted by $MISR_L$ and $MISR_R$ and the Q_L and Q_R sequences are supplied to subsystem B. All components of subsystem B are used along with the precomputed space compaction sequences, M_L^* and M_R^*, to perform data recovery.

A response vector is generated from the sequence of states of an LFSR (discussed in chapter 2) with errors randomly determined according to specified criteria. To generate an $(n \times m)$ response vector consisting of n rows and m columns, the n rows are obtained directly from n successive states of an m-bit LFSR initialized with some non-zero state. By this method an arbitrarily large, good response vector can be generated. To obtain a corresponding faulty response vector, bit errors are introduced in the good vector. The sequence of good and faulty responses comprising a test interval can be generated with successive applications of this technique.

The introduction of errors in a response vector models the behavior of faults in a real circuit. A fault in a circuit can propagate to some maximum number of outputs dependent upon location and input stimulus. Any one stimulus may cause all, some, or none of these outputs to be erroneous. The distribution of errors in response vectors in a test interval adheres to these observations of fault behavior: (1) a fault may affect a maximum set of output data bits, and (2) for any stimulus, some subset of these output data bits may be erroneous.

A single simulation of data recovery of pseudorandomly generated response vectors is determined by a number of parameters supplied to the program, DR. These include the length of the test interval, and the size of a single vector, n and m. Errors within response vectors are randomly determined with the following constraining parameters: (1) the number of data streams in a vector that may contain errors is s, (2) the maximum number of errors in a response vector is t, and (3) the number of response vectors in a test interval containing errors is v.

The process of generating a failing block of 256 response vectors is as follows:

Step 1: Generate 256 good response vectors from $(256 \times n)$ consecutive states of an m-bit LFSR.

Step 2: Randomly select s columns to contain errors $(s = 2)$.

Step 3: Randomly select the set $\mathbf{T}$ of t data bits equally allocated among the previously selected columns $(1 \leq t < s \times n)$.

Step 4: Designate v randomly selected response vectors in the block to contain errors $(1 \leq v \leq 256)$.

Step 5: For each response vector i in the failing block $(1 \leq i \leq 256)$: if it has not been designated to contain errors, response vector i is the good response vector i; else, randomly select a non-empty subset $\mathbf{T}_i$ of $\mathbf{T}$, response vector i is the good response vector i with the data bits in $\mathbf{T}_i$ toggled in value.

Once the block of responses is generated, it is compacted by $MISR_L$ and $MISR_R$ to produce the quotient sequences, Q_L and Q_R. These are supplied to the data recovery subsystem, where error identification is performed. Because the responses were generated pseudorandomly, no faults can be diagnosed and the final simulation step is not performed. The results obtained from this simulation include the number of solutions recovered, the average number of faulty responses considered to eliminate a pair of suspect data streams, and the total time of execution of the error identification algorithms.

The following subsections present the results of the simulations for both overlapping and non-overlapping vectors.

4.2.2 Results for Overlapping Vectors

Several simulations were performed to obtain results for all possible combinations of variables. The experiment variables were: response vector size, number of failing blocks, faulty vectors per block, and percent errors. Four response vector sizes were considered: (128×16), (256×16), (512×16) and (1024×16). For each response

size, data recovery is performed on one and two failing blocks of vectors (block = 256 vectors). The number of faulty response vectors per block is: 8, 16, 32, 64, 128. The percent of bits in error in the failing data streams is: 1.5%, 3%, 6%, 12.5%, 25% and 50%. Each combination of variables is simulated 100 times to obtain 100 random configurations of errors.

The following tables, 4.1 through 4.8, detail the simulation results of the four response vector sizes for one and two failing blocks, respectively. Three quantities are presented for each combination of variables: the average number of solutions obtained, the average number of vectors in a failing block required to eliminate a set of solutions, and the average CPU time required for recovery. Each result is the average of the 100 trials performed. For example, consider table 4.1. For 8 failing vectors in a data block and 1.5% errors in two data streams, the average number of solutions is 21.0, the average number of vectors required to eliminate a set of solutions is 37.3, and the average CPU time is 4.9 seconds.

The following observations can be made from the tables:

1. The number of recovered solutions ranges from approximately 1 to 23.

2. The greatest number of solutions are produced with the fewest percent errors (1.5%) or with the least failing vectors (8).

3. Conversely, the fewest number of solutions are produced with the largest percent errors (50%) and with the most failing vectors (128) (see figure 4.2).

4. The number of solutions tends to decrease with increased percent errors or increased number of failing vectors.

5. Performing recovery on an additional failing block of response vectors produces fewer recovered solutions.

6. The number of vectors considered to reject a set of solutions ranges from approximately 5 to 42.

7. Computation time for response recovery on a 140 Mhz Ultra 1 Sparcstation ranges from approximately 4 seconds for the smallest response size to approximately 130 seconds for the largest.

The simulated response sizes all contained 16 scan chains, therefore the maximum number of solutions is ${}_{16}C_2 = 120$. The greatest average number of solutions is approximately 23 (obtained from an examination of the 8 tables). Thus, on average, nearly 100 possibilities are eliminated, leaving 20 potential solutions. The majority of solutions are eliminated after considering at most 42 vectors, speeding error identification in subsequent vectors. The number of solutions is expected to decrease with the magnitude of the errors: i.e. greater percent errors and numerous failing vectors produce the fewest recovered solutions.

Figure 4.2 is a graphical representation of the data in table 4.7. The X and Y axes are the percentage of errors in the failing vectors and the number of solutions of the recovered data vectors, respectively. Each curve represents the average number of solutions of the 100 simulation runs for a given number of failing vectors indicated by the legend. It can be seen that the number of solutions decreases as the percentage of errors and the number of failing vectors increase.

Although the recovery scheme is relatively inexpensive in terms of computation time, only requiring approximately 2 minutes for the largest response vector size, clearly additional processing is required to reduce the number of potential solutions. The experiment detailed in section 4.3 investigates the feasibility of this approach.

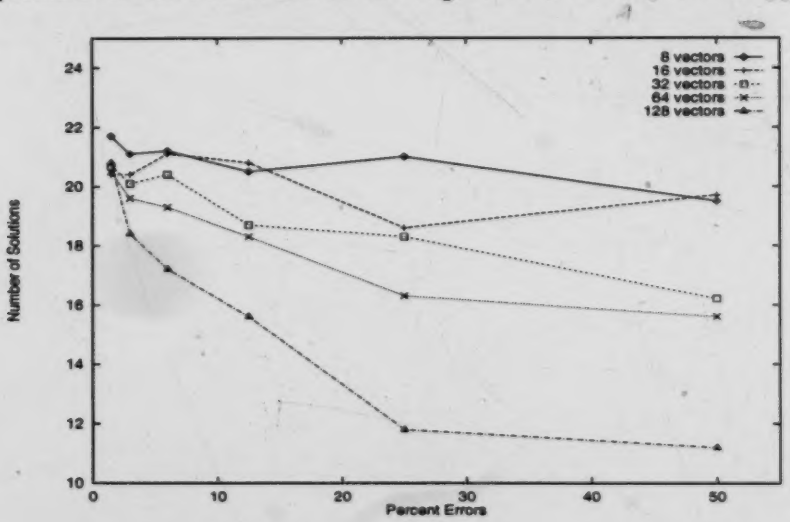

Figure 4.2: Graphical results for vector size 1024 × 16, 1 block

4.2.3 Results for Non-overlapping Vectors

The following tables 4.9 through 4.16 detail the simulation results. The general observations made for the overlapping case in section 4.2.2 hold as well for the non-overlapping approach, with the following significant differences:

1. The number of recovered solutions produced ranges from approximately 1 to 17.
2. The number of vectors considered to reject a set of solutions ranges from approximately 2 to 38.
3. The computation time ranges from 2 seconds for the smallest response vector up to approximately 117 seconds for the largest.

Comparing the overlapping and non-overlapping results clearly shows the superiority of the non-overlapping approach. It produces fewer recovered solutions, approaching one solution as the percent errors increases. The computation time is also reduced as a consequence of more quickly eliminating potential solutions.

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	21.0	20.6	20.4	20.8	20.9	20.2
	Ave. Vectors Considered	37.3	36.4	38.9	35.9	27.8	31.1
	Ave. CPU Time (sec.)	4.9	4.9	5.0	5.0	4.8	5.0
16	Ave. Number Solutions	21.1	21.2	20.0	19.8	18.2	19.5
	Ave. Vectors Considered	19.5	16.5	18.2	15.8	24.1	19.4
	Ave. CPU Time (sec.)	4.3	4.3	4.3	4.3	4.6	4.7
32	Ave. Number Solutions	20.7	20.5	19.6	19.2	18.8	19.4
	Ave. Vectors Considered	11.9	11.4	12.3	12.8	14.7	14.4
	Ave. CPU Time (sec.)	4.1	4.1	4.2	4.3	4.6	4.8
64	Ave. Number Solutions	20.3	20.1	17.6	17.1	16.6	14.5
	Ave. Vectors Considered	6.7	7.5	11.7	11.2	12.4	15.1
	Ave. CPU Time (sec.)	3.9	4.1	4.2	4.4	4.7	4.9
128	Ave. Number Solutions	19.6	18.5	16.5	13.5	13.0	8.7
	Ave. Vectors Considered	7.2	8.0	11.1	15.8	18.9	18.7
	Ave. CPU Time (sec.)	4.0	4.3	4.5	4.7	5.3	5.0

Table 4.1: Results for vector size 128×16, 1 block (overlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	20.2	19.7	20.7	19.6	20.1	20.6
	Ave. Vectors Considered	29.2	35.9	41.8	36.4	37.2	35.2
	Ave. CPU Time (sec.)	6.2	6.4	6.8	6.5	6.8	6.8
16	Ave. Number Solutions	20.2	19.7	18.4	19.0	17.7	18.4
	Ave. Vectors Considered	22.1	20.2	19.6	20.2	21.5	20.3
	Ave. CPU Time (sec.)	6.0	6.0	5.8	6.0	6.0	6.2
32	Ave. Number Solutions	19.8	19.6	19.3	17.9	16.8	15.8
	Ave. Vectors Considered	14.6	12.2	10.1	13.4	17.0	23.1
	Ave. CPU Time (sec.)	5.8	5.8	5.8	5.9	6.1	6.5
64	Ave. Number Solutions	19.4	19.4	16.8	16.9	15.2	14.6
	Ave. Vectors Considered	8.7	6.8	11.2	13.6	22.7	21.3
	Ave. CPU Time (sec.)	5.7	5.8	5.7	6.2	6.8	7.0
128	Ave. Number Solutions	18.7	16.2	16.4	11.9	11.6	8.1
	Ave. Vectors Considered	7.8	13.4	15.4	23.3	28.4	27.7
	Ave. CPU Time (sec.)	5.8	5.9	6.5	6.4	7.3	6.9

Table 4.2: Results for vector size 128×16, 2 blocks (overlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	20.4	20.5	20.2	20.5	20.1	20.2
	Ave. Vectors Considered	31.7	36.5	36.3	32.3	31.3	32.1
	Ave. CPU Time (sec.)	11.3	11.6	11.7	11.6	11.6	11.6
16	Ave. Number Solutions	20.9	20.8	19.6	19.0	19.5	19.2
	Ave. Vectors Considered	16.3	22.4	18.9	17.1	18.6	16.4
	Ave. CPU Time (sec.)	10.5	10.9	10.6	10.6	10.9	10.8
32	Ave. Number Solutions	18.9	19.1	17.4	18.3	17.9	17.5
	Ave. Vectors Considered	10.0	10.1	12.2	12.3	10.4	15.5
	Ave. CPU Time (sec.)	9.9	10.0	10.0	10.4	10.4	11.0
64	Ave. Number Solutions	18.6	17.7	18.0	16.8	15.9	16.2
	Ave. Vectors Considered	7.0	8.0	8.0	9.9	11.6	8.9
	Ave. CPU Time (sec.)	9.8	9.9	10.2	10.5	10.9	11.1
128	Ave. Number Solutions	18.4	17.8	17.8	15.8	10.9	9.0
	Ave. Vectors Considered	7.0	7.5	11.2	14.2	18.3	17.3
	Ave. CPU Time (sec.)	10.0	10.2	11.1	11.6	11.6	11.7

Table 4.3: Results for vector size 256×16, 1 block (overlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	22.0	21.4	22.5	19.6	19.4	21.4
	Ave. Vectors Considered	35.9	32.2	26.8	37.8	41.0	30.2
	Ave. CPU Time (sec.)	15.2	14.9	15.1	15.0	15.5	15.7
16	Ave. Number Solutions	21.4	22.0	21.6	20.1	19.2	19.3
	Ave. Vectors Considered	19.6	17.7	17.2	23.5	22.4	20.9
	Ave. CPU Time (sec.)	14.1	14.3	14.4	14.6	14.6	15.0
32	Ave. Number Solutions	21.3	19.6	21.6	20.6	19.3	18.3
	Ave. Vectors Considered	8.5	13.1	12.3	11.9	18.9	18.9
	Ave. CPU Time (sec.)	13.6	13.5	14.4	14.5	15.3	15.6
64	Ave. Number Solutions	20.5	20.8	20.6	17.2	15.6	13.1
	Ave. Vectors Considered	6.3	8.5	13.3	15.6	19.9	24.6
	Ave. CPU Time (sec.)	13.5	14.1	14.9	14.6	15.3	15.8
128	Ave. Number Solutions	19.5	17.8	16.4	14.6	11.1	9.7
	Ave. Vectors Considered	8.9	10.8	16.7	20.5	32.0	24.4
	Ave. CPU Time (sec.)	14.0	14.0	14.9	15.6	16.7	16.5

Table 4.4: Results for vector size 256×16, 2 blocks (overlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	21.8	20.9	21.0	20.9	20.4	19.5
	Ave. Vectors Considered	32.0	32.8	30.3	32.6	30.3	27.2
	Ave. CPU Time (sec.)	33.6	34.0	33.8	34.3	34.0	33.5
16	Ave. Number Solutions	21.2	21.9	20.3	19.9	20.0	18.9
	Ave. Vectors Considered	18.4	14.5	15.7	19.9	20.3	21.3
	Ave. CPU Time (sec.)	31.8	32.0	31.9	32.6	33.0	33.1
32	Ave. Number Solutions	20.3	19.3	20.0	18.7	16.8	17.0
	Ave. Vectors Considered	9.6	10.8	12.1	13.5	15.7	13.4
	Ave. CPU Time (sec.)	30.5	30.9	31.7	31.9	32.1	32.4
64	Ave. Number Solutions	21.1	20.1	18.3	17.2	14.8	14.4
	Ave. Vectors Considered	8.1	8.6	9.4	10.6	14.1	14.9
	Ave. CPU Time (sec.)	30.8	31.4	31.4	32.0	32.5	33.6
128	Ave. Number Solutions	20.3	20.2	14.6	15.5	10.6	8.9
	Ave. Vectors Considered	6.8	6.4	9.9	17.2	18.1	21.8
	Ave. CPU Time (sec.)	30.9	31.9	31.3	34.5	33.7	35.8

Table 4.5: Results for vector size 512×16, 1 block (overlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	21.6	20.7	21.2	20.8	20.5	20.8
	Ave. Vectors Considered	35.2	30.7	31.7	33.2	30.3	36.2
	Ave. CPU Time (sec.)	42.6	40.6	41.3	41.5	41.2	42.3
16	Ave. Number Solutions	21.2	21.6	20.4	21.6	20.2	18.9
	Ave. Vectors Considered	17.1	17.6	17.7	18.6	22.0	24.0
	Ave. CPU Time (sec.)	40.1	39.7	39.2	40.6	40.6	40.9
32	Ave. Number Solutions	21.5	20.4	19.6	19.8	18.2	18.6
	Ave. Vectors Considered	8.9	10.3	17.2	12.2	16.2	15.6
	Ave. CPU Time (sec.)	42.2	38.4	39.4	39.5	40.0	41.6
64	Ave. Number Solutions	20.8	20.3	18.7	16.6	15.1	13.8
	Ave. Vectors Considered	8.7	9.5	12.9	13.7	20.4	19.0
	Ave. CPU Time (sec.)	39.5	39.0	39.4	39.2	40.9	41.5
128	Ave. Number Solutions	19.8	19.0	16.7	14.4	12.7	9.3
	Ave. Vectors Considered	8.6	8.6	11.8	20.2	28.1	31.2
	Ave. CPU Time (sec.)	39.8	39.5	39.7	41.7	44.6	45.2

Table 4.6: Results for vector size 512×16, 2 blocks (overlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	21.7	21.1	21.2	20.5	21.0	19.5
	Ave. Vectors Considered	30.6	37.9	31.1	31.3	30.8	36.3
	Ave. CPU Time (sec.)	109.5	111.5	109.5	109.5	111.1	111.5
16	Ave. Number Solutions	20.4	20.4	21.1	20.8	18.6	19.7
	Ave. Vectors Considered	16.5	18.7	19.5	22.3	19.1	18.4
	Ave. CPU Time (sec.)	105.3	105.7	106.7	108.0	106.9	108.3
32	Ave. Number Solutions	20.7	20.1	20.4	18.7	18.3	16.2
	Ave. Vectors Considered	9.4	9.8	11.3	15.5	14.4	15.2
	Ave. CPU Time (sec.)	103.8	104.0	105.6	106.5	107.3	109.6
64	Ave. Number Solutions	20.5	19.6	19.3	18.3	16.3	15.6
	Ave. Vectors Considered	6.6	7.2	9.6	11.6	14.6	12.1
	Ave. CPU Time (sec.)	103.5	103.8	107.6	107.0	108.3	109.9
128	Ave. Number Solutions	20.8	18.4	17.2	15.6	11.8	11.2
	Ave. Vectors Considered	5.6	8.1	10.6	12.7	18.4	17.8
	Ave. CPU Time (sec.)	104.5	104.8	106.6	108.3	112.6	119.4

Table 4.7: Results for vector size 1024×16, 1 block (overlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	20.4	20.8	20.7	20.1	20.5	19.7
	Ave. Vectors Considered	32.0	26.5	29.6	29.9	25.7	30.9
	Ave. CPU Time (sec.)	122.1	123.0	123.9	122.4	122.9	124.9
16	Ave. Number Solutions	20.9	20.7	20.5	21.1	19.5	19.2
	Ave. Vectors Considered	18.8	19.2	21.2	18.6	22.7	19.1
	Ave. CPU Time (sec.)	119.6	120.8	120.7	121.8	122.1	122.6
32	Ave. Number Solutions	20.8	21.2	19.8	18.2	18.6	17.3
	Ave. Vectors Considered	9.8	9.0	13.0	15.4	14.3	18.4
	Ave. CPU Time (sec.)	117.6	118.7	119.0	119.0	121.4	123.5
64	Ave. Number Solutions	22.3	19.1	19.1	16.3	16.4	13.3
	Ave. Vectors Considered	7.4	11.4	9.8	17.6	17.0	16.7
	Ave. CPU Time (sec.)	119.9	118.2	119.5	120.3	124.0	125.8
128	Ave. Number Solutions	20.0	17.8	18.3	14.4	11.1	8.6
	Ave. Vectors Considered	5.2	11.7	14.6	21.8	29.9	23.4
	Ave. CPU Time (sec.)	117.9	119.4	124.6	125.2	128.0	129.6

Table 4.8: Results for vector size 1024×16, 2 blocks (overlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	16.7	13.6	8.9	3.8	2.0	1.4
	Ave. Vectors Considered	36.2	34.2	31.5	29.1	31.3	28.7
	Ave. CPU Time (sec.)	4.2	3.9	3.5	3.1	3.1	3.0
16	Ave. Number Solutions	15.4	12.3	7.3	4.7	2.7	1.2
	Ave. Vectors Considered	16.7	17.0	18.7	15.1	19.1	15.3
	Ave. CPU Time (sec.)	3.5	3.3	3.0	2.7	2.7	2.5
32	Ave. Number Solutions	16.4	12.4	7.9	4.7	2.2	1.3
	Ave. Vectors Considered	10.3	8.5	7.9	8.6	9.3	7.7
	Ave. CPU Time (sec.)	3.4	3.0	2.7	2.5	2.3	2.2
64	Ave. Number Solutions	15.5	12.0	7.7	5.3	2.2	1.3
	Ave. Vectors Considered	4.6	4.8	4.4	4.7	4.4	3.3
	Ave. CPU Time (sec.)	3.2	2.9	2.5	2.4	2.2	2.1
128	Ave. Number Solutions	16.1	11.9	9.3	4.8	2.3	1.3
	Ave. Vectors Considered	2.4	2.5	2.2	2.1	2.2	2.4
	Ave. CPU Time (sec.)	3.3	2.9	2.7	2.3	2.2	2.2

Table 4.9: Results for vector size 128×16, 1 block (nonoverlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	16.6	12.1	8.0	4.0	2.3	1.3
	Ave. Vectors Considered	30.8	33.8	37.0	33.2	34.8	28.7
	Ave. CPU Time (sec.)	5.4	4.8	4.3	3.6	3.5	3.2
16	Ave. Number Solutions	17.5	13.0	8.8	4.6	2.4	1.3
	Ave. Vectors Considered	16.2	16.5	15.0	18.2	16.0	16.0
	Ave. CPU Time (sec.)	5.1	4.4	3.7	3.2	2.9	2.8
32	Ave. Number Solutions	17.1	13.1	8.3	4.7	2.4	1.2
	Ave. Vectors Considered	8.7	8.7	9.4	9.2	8.2	8.4
	Ave. CPU Time (sec.)	4.8	4.2	3.5	3.0	2.6	2.5
64	Ave. Number Solutions	15.6	12.8	7.8	4.7	2.7	1.3
	Ave. Vectors Considered	4.4	4.4	5.2	4.4	4.0	3.5
	Ave. CPU Time (sec.)	4.5	4.1	3.4	2.9	2.6	2.4
128	Ave. Number Solutions	16.1	13.3	7.2	4.7	2.6	1.4
	Ave. Vectors Considered	2.5	2.5	2.5	2.2	2.2	1.8
	Ave. CPU Time (sec.)	4.7	4.3	3.3	2.9	2.6	2.5

Table 4.10: Results for vector size 128×16, 2 blocks (nonoverlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	15.8	12.4	7.8	6.0	2.4	1.4
	Ave. Vectors Considered	32.0	30.4	30.8	35.3	33.9	28.7
	Ave. CPU Time (sec.)	10.6	9.6	9.1	9.2	8.8	8.2
16	Ave. Number Solutions	16.2	13.4	9.5	4.4	1.9	1.2
	Ave. Vectors Considered	16.1	16.0	16.2	19.0	15.8	16.3
	Ave. CPU Time (sec.)	9.7	8.9	8.4	7.9	7.5	7.4
32	Ave. Number Solutions	15.8	12.6	8.8	5.2	2.1	1.2
	Ave. Vectors Considered	9.8	9.5	8.6	9.8	9.4	7.2
	Ave. CPU Time (sec.)	9.3	8.4	7.9	7.5	7.1	6.8
64	Ave. Number Solutions	16.6	13.2	8.2	4.6	2.0	1.2
	Ave. Vectors Considered	4.3	5.0	4.1	4.1	4.6	3.9
	Ave. CPU Time (sec.)	9.1	8.3	7.6	7.0	6.8	6.6
128	Ave. Number Solutions	15.7	11.6	9.2	5.0	2.1	1.2
	Ave. Vectors Considered	2.6	2.2	2.4	2.2	2.2	2.1
	Ave. CPU Time (sec.)	9.0	8.0	7.8	7.2	6.7	6.8

Table 4.11: Results for vector size 256×16, 1 block (nonoverlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	15.2	12.5	8.5	5.3	2.5	1.2
	Ave. Vectors Considered	31.7	31.6	29.6	37.7	28.0	26.3
	Ave. CPU Time (sec.)	12.5	11.6	10.4	10.1	8.8	8.6
16	Ave. Number Solutions	16.2	13.2	8.6	4.6	2.2	1.3
	Ave. Vectors Considered	14.5	14.4	17.0	15.4	18.8	14.5
	Ave. CPU Time (sec.)	11.8	10.7	9.6	8.4	8.2	7.8
32	Ave. Number Solutions	15.3	12.3	8.0	4.0	2.0	1.2
	Ave. Vectors Considered	8.2	7.9	9.8	9.7	7.1	7.0
	Ave. CPU Time (sec.)	11.2	10.2	9.1	7.9	7.4	7.4
64	Ave. Number Solutions	15.6	12.6	10.0	4.1	2.3	1.1
	Ave. Vectors Considered	4.0	4.7	4.4	4.6	4.1	3.5
	Ave. CPU Time (sec.)	11.2	10.2	9.6	7.7	7.4	7.4
128	Ave. Number Solutions	16.7	11.4	6.8	5.2	1.9	1.1
	Ave. Vectors Considered	2.3	2.5	2.0	2.6	2.5	2.2
	Ave. CPU Time (sec.)	11.8	10.1	8.6	8.3	7.4	7.7

Table 4.12: Results for vector size 256×16, 2 blocks (nonoverlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	15.3	12.8	8.8	5.1	2.2	1.1
	Ave. Vectors Considered	26.9	27.5	30.5	33.1	31.1	33.3
	Ave. CPU Time (sec.)	30.5	29.6	29.0	28.4	27.5	27.6
16	Ave. Number Solutions	15.8	13.4	9.4	5.0	2.0	1.4
	Ave. Vectors Considered	17.3	18.7	21.5	18.1	15.2	20.0
	Ave. CPU Time (sec.)	29.5	28.6	27.9	26.3	25.1	25.8
32	Ave. Number Solutions	16.2	13.8	8.8	4.9	2.3	1.2
	Ave. Vectors Considered	8.9	8.6	8.0	7.4	8.8	8.5
	Ave. CPU Time (sec.)	28.5	27.5	25.9	24.7	24.3	24.5
64	Ave. Number Solutions	15.1	13.0	8.1	4.3	1.5	1.2
	Ave. Vectors Considered	4.3	4.2	4.0	4.4	4.1	3.7
	Ave. CPU Time (sec.)	27.7	27.1	25.5	24.6	23.9	24.1
128	Ave. Number Solutions	16.3	13.0	8.4	4.1	1.9	1.2
	Ave. Vectors Considered	2.1	2.2	2.2	2.1	2.0	1.9
	Ave. CPU Time (sec.)	28.1	27.3	25.8	24.5	24.0	24.5

Table 4.13: Results for vector size 512×16, 1 block (nonoverlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	17.1	13.4	8.7	5.0	2.3	1.3
	Ave. Vectors Considered	29.4	29.2	25.5	30.1	25.4	35.7
	Ave. CPU Time (sec.)	37.6	34.8	31.5	29.9	27.7	28.8
16	Ave. Number Solutions	17.9	13.2	8.3	4.7	1.9	1.3
	Ave. Vectors Considered	15.0	13.6	17.0	16.3	15.5	14.4
	Ave. CPU Time (sec.)	36.2	32.6	30.2	27.7	26.2	26.0
32	Ave. Number Solutions	16.7	14.3	9.2	5.6	2.1	1.4
	Ave. Vectors Considered	8.7	8.2	8.8	9.5	8.7	7.3
	Ave. CPU Time (sec.)	34.8	32.8	29.7	27.6	25.4	25.3
64	Ave. Number Solutions	16.9	13.1	9.5	4.5	2.2	1.3
	Ave. Vectors Considered	4.0	4.5	4.1	4.3	4.5	4.0
	Ave. CPU Time (sec.)	34.6	31.9	29.7	26.4	25.1	25.5
128	Ave. Number Solutions	17.4	13.3	9.5	4.3	1.9	1.2
	Ave. Vectors Considered	2.2	2.5	2.1	2.3	1.8	2.2
	Ave. CPU Time (sec.)	35.3	32.4	30.1	26.5	24.9	26.2

Table 4.14: Results for vector size 512×16, 2 blocks (nonoverlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	14.7	14.0	8.5	4.7	2.3	1.2
	Ave. Vectors Considered	27.7	33.3	32.6	31.3	35.8	27.6
	Ave. CPU Time (sec.)	102.4	103.4	100.3	98.1	98.5	96.2
16	Ave. Number Solutions	16.6	14.0	8.8	4.3	2.1	1.1
	Ave. Vectors Considered	15.6	14.3	20.0	15.5	14.8	14.6
	Ave. CPU Time (sec.)	100.4	98.3	96.8	93.2	92.4	92.5
32	Ave. Number Solutions	16.5	12.1	8.7	5.0	1.7	1.1
	Ave. Vectors Considered	8.0	8.9	9.0	9.6	7.9	8.2
	Ave. CPU Time (sec.)	98.4	95.7	93.7	92.0	89.7	91.3
64	Ave. Number Solutions	17.4	12.4	7.2	4.7	2.3	1.2
	Ave. Vectors Considered	4.4	4.6	4.0	4.0	3.9	4.3
	Ave. CPU Time (sec.)	98.4	95.3	92.1	90.7	90.0	95.6
128	Ave. Number Solutions	17.0	12.7	9.2	4.7	2.1	1.2
	Ave. Vectors Considered	2.3	2.2	2.2	2.3	2.1	1.9
	Ave. CPU Time (sec.)	98.1	95.3	93.5	91.6	89.9	97.0

Table 4.15: Results for vector size 1024×16, 1 block (nonoverlapped vectors)

Failing Vectors	Percent Errors	1.5%	3%	6%	12.5%	25%	50%
8	Ave. Number Solutions	17.4	13.2	9.2	4.5	2.2	1.2
	Ave. Vectors Considered	32.9	33.5	33.8	34.5	33.2	30.1
	Ave. CPU Time (sec.)	116.6	111.8	106.9	101.6	99.3	97.6
16	Ave. Number Solutions	17.3	13.0	7.2	4.6	2.1	1.3
	Ave. Vectors Considered	15.5	14.7	18.7	19.4	14.8	14.0
	Ave. CPU Time (sec.)	112.0	106.1	100.0	97.5	93.1	93.3
32	Ave. Number Solutions	17.4	13.2	8.9	4.9	1.7	1.2
	Ave. Vectors Considered	7.5	8.9	10.1	9.1	8.3	7.4
	Ave. CPU Time (sec.)	110.4	105.1	100.0	94.8	91.0	92.2
64	Ave. Number Solutions	16.3	13.2	8.3	4.5	1.9	1.1
	Ave. Vectors Considered	4.6	4.3	3.9	4.5	4.4	3.6
	Ave. CPU Time (sec.)	108.9	104.5	98.1	93.6	90.6	93.2
128	Ave. Number Solutions	17.0	13.7	7.5	4.3	2.6	1.2
	Ave. Vectors Considered	2.1	1.9	2.2	2.2	2.1	1.9
	Ave. CPU Time (sec.)	110.5	106.0	97.7	93.6	92.1	98.3

Table 4.16: Results for vector size 1024×16, 2 blocks (nonoverlapped vectors)

4.3 Data Recovery in Benchmark Circuits

The goal of the second set of simulations is to demonstrate that multiple solutions produced by algorithm 3.4.1 can be further eliminated by structural analysis and fault simulation of the circuit. To that end, benchmark circuits are used to obtain faulty responses by fault simulation of stuck-at faults. Data recovery is thus performed on faulty responses of actual circuit faults. The number of solutions can then be reduced by structural analysis and fault simulation.

4.3.1 Simulation Environment

Referring again to figure 4.1, the system components used in this simulation are as follows. Within subsystem A, response vectors are created by the *logic simulator*. It simulates faults within the *benchmark circuits*, stimulated by pseudorandom test patterns obtained from the *LFSR*. As in the first simulation, the generated response vectors are then compacted by $MISR_L$ and $MISR_R$ and the Q_L and Q_R sequences are supplied to subsystem B. All components of subsystem B are likewise used along with the precomputed space compaction sequences, M_L^* and M_R^*, to perform data recovery. After data recovery, structural analysis and fault simulation are performed to reduce any multiple recovered solutions (not shown in figure 4.1).

The circuits used for this simulation are the ISCAS85 suite of benchmarks [7]. The ISCAS85 benchmarks have been the standard benchmark circuits used to evaluate, among other things, testing and simulation algorithms. Table 4.17 provides the details of the benchmark circuits in the ISCAS85 suite along with the number of stuck-at fault classes simulated.

In order to simulate a STUMPS-like architecture with small combinational circuits (to apply the data recovery scheme), circuit outputs are partitioned into distinct data streams. A circuit with 32 primary outputs can be viewed as ($m = 4$) data streams each of length ($n = 8$). The first data stream consists of outputs (1 - 8), the second consists of outputs (9 - 16), the third consists of outputs (17 - 24), and the fourth consists of outputs (25 - 32). Outputs in data streams are arranged and compacted in numerical order: the data bits from outputs (1, 9, 17, 25) are compacted first, followed by (2, 10, 18, 26), (3, 11, 19, 27), until lastly (8, 16, 24, 32) are compacted.

The data recovery scheme can then be applied without having STUMPS-equipped benchmark circuits. Table 4.18 provides the partitioning of outputs used for each benchmark circuit in terms of n and m, along with the simulation results.

Unlike STUMPS for sequential circuits, where the data streams serve as both the inputs and the outputs of the combinational logic blocks, the benchmark circuits have an unequal number of inputs and outputs. For simplicity, test patterns are supplied by an LFSR of length equal to the number of inputs, connected directly to the circuit inputs. Thus for benchmark circuit *c1355*, a 41-bit LFSR is used to generate test patterns, with the least significant bit of the LFSR connected to the first input of the circuit.

Name	Function	Inputs	Outputs	Gates
c1355	ECAT	41	32	546
c1908	ECAT	33	25	880
c432	Priority Decoder	36	7	160
c499	ECAT	41	32	202
c880	ALU and Control	60	26	383
c2670	ALU and Control	233	140	1193
c3540	ALU and Control	50	22	1669
c5315	ALU and Selector	178	123	2307
c6288	16-bit Multiplier	32	32	2416
c7552	ALU and Control	207	108	3512

Table 4.17: Characteristics of ISCAS85 benchmark circuits

A failing block of responses is obtained by fault simulating single stuck-at faults in the benchmark circuits. The faults chosen are representative members of fault classes established by fault collapsing. Further, only faults activated within 64K test patterns are considered; faults not activated within this test length are removed from consideration as they do not produce faulty responses within the test length. The responses in the failing block are then compacted by $MISR_L$ and $MISR_R$, according to the output partitioning scheme described above, to generated Q_L and Q_R. As in the first simulation, Q_L and Q_R are supplied to the data recovery subsystem. Additionally, an equal length good circuit simulation is performed to obtain the reference

space compaction sequences, M_L^* and M_R^*.

The program parameters that determine a simulation include: the benchmark circuit to simulate, the partition of the circuit outputs, the list of faults to simulate, and the test length to consider. The simulation results consist of: the percent of faults that are recoverable, the percent of data recovery results that are multiple solutions, the average number of multiple solutions, the percent of fault simulation results that produce multiple solutions, and the time of execution per fault.

4.3.2 Results

Table 4.18 summarizes the results of the second experiment for one failing block of 256 response vectors. The rows of the table correspond to the 10 benchmark circuits in the ISCAS85 suite. From left to right, each column lists the following, with the column number indicated in parentheses: the benchmark circuit name (1); the circuit output partitioning scheme used, in terms of m (2) and n (3); the number of collapsed stuck-at faults (4); the number of recoverable faults (i.e. faults that affect at most two scan chains) (5); the percent of total faults that are recoverable (6); the percent of data recovery results that are multiple solutions (7); the average number of multiple solutions recovered (8); the percent of fault simulation results that produce multiple solutions (9); and the average recovery/fault simulation time required per fault (10). For example, consider table 4.18. Reading across the row corresponding to circuit *c1355*, the circuit outputs are partitioned into 4 data streams of length 8 (this corresponds to the 32 outputs given for *c1355* in table 4.17); the number of faults considered is 1566; of these faults, 954 or 60.9% of the total faults are recoverable; 96.0% of the results after data recovery are multiple solutions; the average number of multiple solutions is 2.70; 0% of the results after fault simulation are multiple solutions; and the average time required per fault is 1.95 seconds.

Three significant observations can be made from the data:

1. A large percent of the simulated faults affect two or fewer scan chains (ranging from approximately 57% for the smaller circuits up to 90% or more for the larger circuits).

2. The average number of multiple solutions ranges from approximately 2 to 5 solutions, discounting the two benchmark circuits, *c432* and *c3540*, that did not produce multiple solutions.

3. Nearly all multiple solutions are resolved after fault simulation except for circuit *c5315* where only 2.0% of the results after fault simulation were multiple solutions.

The following conclusions can be drawn from the simulation of the data recovery scheme on standard benchmark circuits. The data recovery scheme is applicable to a large percent of the modeled stuck-at faults (90% or more for the larger benchmark circuits). The data recovery scheme reduces the average number of multiple solutions by at least 60% of the maximum number possible, ${}_mC_2$. And lastly, post-data recovery fault simulation is a feasible method of further reducing multiple solutions to a single solution.

1	2	3	4	5	6	7	8	9	10
Circuit Name	m	n	Total Faults	Recoverable Faults	Percent Recoverable Faults	Multiple Recovered Solutions	Average Multiple Solutions	Multiple Solutions after Fault Simulation	Time per Fault
c1355	4	8	1566	954	60.9%	96.0%	2.70	0%	1.95s
c1908	3	9	1884	1332	70.7%	68.2%	2.02	0%	0.91s
c432	2	4	500	500	100%	0%	0	0%	0.14s
c499	4	8	718	408	56.8%	93.1%	2.69	0%	0.65s
c880	4	7	982	981	99.9%	96.3%	2.51	0%	0.74s
c2670	5	28	2081	2081	100%	98.4%	4.66	0%	5.98s
c3540	2	11	3304	3304	100%	0%	0	0%	0.32s
c5315	5	25	5298	5133	96.9%	97.0%	4.21	2.0%	9.13s
c6288	4	8	7710	7480	97.0%	95.9%	2.73	0%	3.22s
c7552	6	18	7249	6588	90.9%	97.8%	5.09	0%	12.52s

Table 4.18: Results of data recovery in standard benchmark circuits

4.4 Software Implementation

This section discusses specific implementation details of the software system, *DR*, used to test and verify the data recovery scheme. DR is a collection of software modules

that perform logic simulation, fault simulation, creation of pseudorandomly generated responses, output response compaction, data construction, error identification and circuit structural analysis. It functions in two distinct modes: (1) data recovery of pseudorandomly generated faulty responses, and (2) data recovery in standard benchmark circuits. To support mode (2), DR contains a two-valued parallel logic simulator capable of computing the circuit response to 256 test vectors in parallel [19].

The DR software system used to implement the error identification algorithms and test the data recovery schemes was written in C++. The various modules of the system were implemented as C++ objects, or *classes*. A C++ class is a collection data and functions that operate on the data. These can be logically divided into three categories: circuit-related, simulation-related and miscellaneous classes. Circuit-related classes are responsible for the creation and the representation of a logic circuit from an external format. Simulation-related classes implement logic and fault simulation, response compaction, error identification and circuit structural analysis. The miscellaneous classes perform low level functions used by other modules.

Circuit-Related Modules

Node Class

The Node class abstracts the interconnections between logic gates in a digital circuit. All Nodes, but for the primary input and primary output Nodes, must have a source Gate and at least one fanout Gate. The primary input Nodes have no source Gate, while the primary output Nodes have no fanout Gate. The member variables of a Node object are: a unique numerical address, the Node name, the source Gate, the number of fanout Gates, an array of Gates in the fanout, and an Int256 object (see below) which represents the 256 logic values of the Node when simulated in parallel. Figure 4.3 illustrates the basic structure of a Node object.

To allow quick access to any specific Node, all Nodes within the circuit are stored in a DynamicArray structure (see below), indexed by the Node address. This structure grows in size as the circuit is being created and new Nodes are added while permitting direct access to individual Nodes like a standard array.

Gate Class

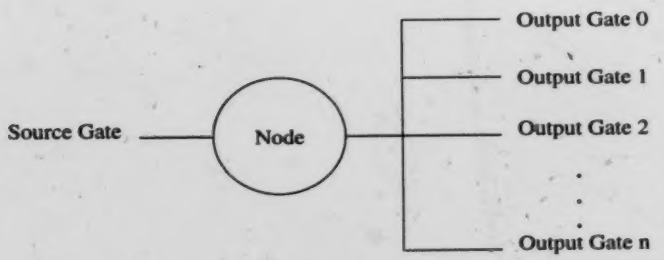

Figure 4.3: Node object

The Gate class represents a single logic gate in a circuit. It is an abstract class which implies that no object of type Gate can exist. Instead, Gate serves as a base class from which subclasses representing the individual logic gates are derived; figure 4.4 illustrates the Gate class hierarchy.

The member variables of the Gate class include the number of inputs, the output Node, and an array of input Nodes. Each subclass derived from the base has a specific *Evaluate()* member function that implements the behavior of the respective logic gate. All Gate objects in the circuit are stored in a global singly linked-list. Figure 4.5 illustrates the basic structure of a Gate object.

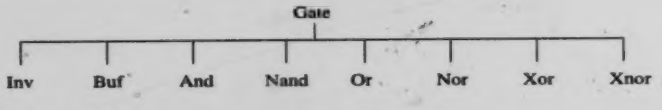

Figure 4.4: Gate class hierarchy

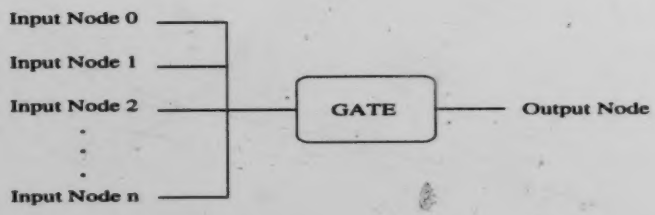

Figure 4.5: Gate object

Int256 Class

The Int256 class represents the logic values of circuit nodes. The implemented logic simulator emulates the parallel logic simulator in [19]. It simulates 256 circuit re-

sponses concurrently. The Int256 class supports this behavior by representing a collection of 256 logic values in a 256-bit array. Operator overloading is used to define logic operators on Int256 objects. The standard logic operators are defined, including AND, OR, NOT and XOR, as well as the assignment operator.

read_iscas85() Function

The read_iscas85() function parses an ISCAS85 net-list file and creates the appropriate Gate and Node objects to represent the circuit. The net-list file is parsed one line at a time; the logic gate is extracted and the respective Gate and Node objects are created. Extensive use of regular expressions in the **g++** built-in String class are used to parse each circuit line. Error-checking is performed on each line read to ensure a valid ISCAS85 circuit is created.

Simulation-Related Modules

Element Class

The Element class is an elementary structure used to contain information describing an element of the data recovery scheme. It contains row, column and vector member variables to uniquely specify an element. Equality and inequality operators are also defined to facilitate comparison of Element objects.

Fault Class

The Fault class encapsulates an individual stuck-at fault. The member variables include: the type of stuck-at fault (input or output); the fault location in terms of a Gate and Node object; and the stuck-at value (either logic 1 or logic 0). Faults can be contained in FaultList objects, implemented as singly linked-lists. Also, fault collapsing employing two-way equivalence [2] is implemented as a FaultList member function.

Register Class

The Register class implements the MISR and LFSR objects used in the simulation. Register is the base class, and the MISR and LFSR classes are derived from the base class. To support large registers (more than 32 bits), the actual shift register is implemented with **g++** built-in Integer objects. The Integer class provides multiple precision integer arithmetic, including logic operations. Thus, effectively any length

of shift register can be realized.

ErrorId Class

The ErrorId class implements the two error identification algorithms presented in chapter 3, algorithm 3.3.1 and algorithm 3.4.1. ErrorId is utilized by the Sim class to perform error identification. Solving the generated systems of equations is accomplished by Gaussian elimination and back-substitution.

Sim Class

The Sim class implements the two basic simulations performed in DR: error identification in pseudorandomly generated responses (section 4.2); and fault simulation in benchmark circuits (section 4.3). To support these simulations, the Sim class also includes parallel logic simulation, fault simulation, pseudorandom response generation, and data construction.

Miscellaneous Modules

SparseMatrix Class

The SparseMatrix class implements sparse matrices [18] and corresponding operations used to solve systems of Boolean equations. The equations to be solved contain at most two unknowns, thus the resulting augmented matrix of the system has a large proportion of zero terms. To decrease memory usage and speed up matrix operations only the non-zero terms are stored explicitly in the matrix. Further, since only Boolean equations are considered, the value of the non-zero terms is always 1 and is not stored explicitly.

A $(n \times n)$ SparseMatrix is represented by n singly linked-lists, one for each row in the matrix. Each linked-list stores the indices of the non-zero terms in the respective row, sorted in ascending order. Figure 4.6 illustrates the basic structure of a SparseMatrix object. The multiplication of row elements as part of Gaussian elimination is optimized to only consider the non-zero matrix elements.

DynamicArray Template

The DynamicArray template is a parameterized container class. It implements a generic dynamic array for programmer-specified objects. Using templates permits the re-use of the same dynamic array code for any object. Thus dynamic arrays of

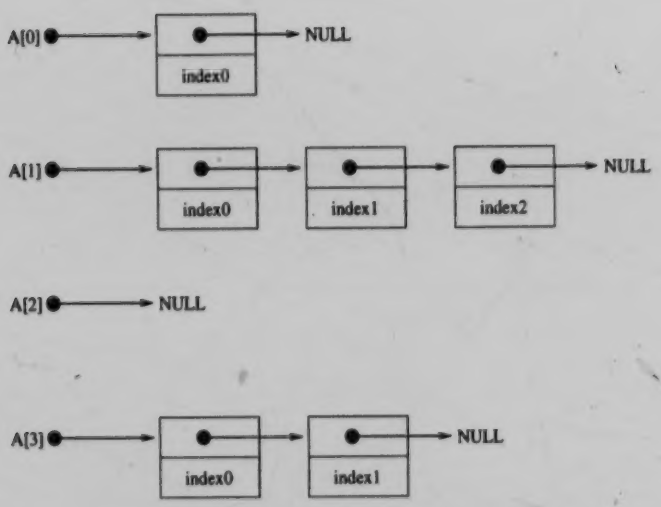

Figure 4.6: A sample SparseMatrix $A[][]$

integers, characters or pointers can be easily created. DynamicArrays are used to contain pointers to Node objects. Nodes are created dynamically as an ISCAS85 netlist file is parsed, thus occupying an unknown amount of memory. Direct access to individual Nodes is required to build the interconnections between the new Node and the partial circuit in memory. Using linked-lists was rejected because of the slower access to individual list elements.

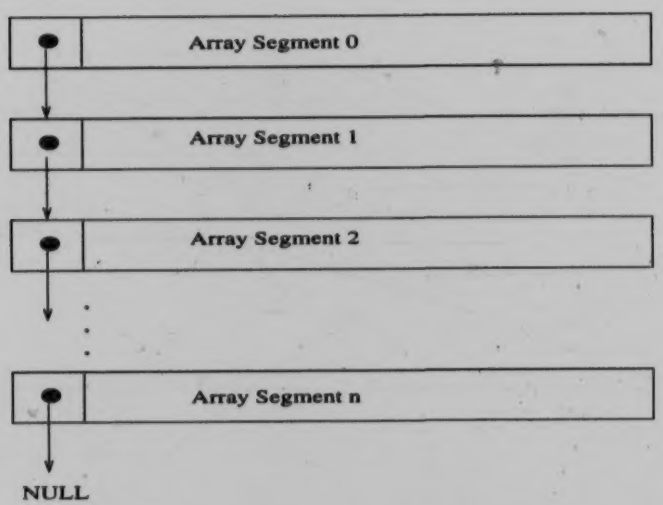

Figure 4.7: The DynamicArray structure

Chapter 5

Conclusion

Fault diagnosis in a *built-in self test* (BIST) environment presents unique challenges not found in external testing. In external testing, the circuit response information is readily available for capture by the tester. In BIST, however, the circuit responses are compacted and lost during testing. Methods to retrieve the lost information are of importance to fault diagnosis. By retrieving the uncompacted circuit responses the observability of the *circuit under test* (CUT) is improved, enhancing the diagnostic resolution and resulting in a more specific analysis of the failure.

A well known BIST architecture used for *integrated circuit* (IC) testing is *self-test using MISR/parallel SRSG*[1] (STUMPS) [5, 12]. STUMPS presents many internal circuit observation points from which data is compacted towards a final signature. The application of test patterns and circuit response compaction occurs completely on-chip. Only the final signature is extracted from the CUT where it can be compared by the tester with an error-free reference signature. The original circuit responses are thus normally unavailable for fault diagnosis.

The diagnosis method used in the STUMPS environment [19, 20] employs data retrieval to obtain the lost circuit responses from the CUT. Intermediate signatures are used to target failing blocks of circuit responses. All circuit responses in targeted blocks are scanned out of the CUT and stored, to be used for fault diagnosis off-line. The data retrieval process is time consuming and transfers a large volume of data from the CUT to the tester. Fault diagnosis can be performed almost equally well

[1] Acronym of shift-register sequence generator.

based on less data transferred from the CUT, speeding up data retrieval and reducing tester usage.

This thesis presents a new primary data recovery scheme for diagnosis in a STUMPS testing environment. We model MISR-based data compaction as separate space and time compactions, then present a method for constructing a space compaction sequence from the quotient sequence and the signature obtained from the MISR. Errors in the circuit response data correspond to specific errors in the space compaction sequence, whereas no such correspondence exists in the quotient sequence (all bits following the introduction of the error can be perturbed). Errors in the circuit responses can then be identified by solving a system of equations based on the space compaction sequences. The identified errors are combined with the good circuit responses, obtained by simulation, to produce the uncompacted circuit responses of the CUT. The new method reduces the volume of data transferred from the CUT to the tester by a factor of $\frac{2}{m}$ (where m is the number of scan chains in STUMPS), while achieving comparable recovery of the lost circuit response data.

Two recommendations are given to improve the effectiveness of the primary data recovery scheme. The first recommendation uses non-overlapping response vectors to enhance error identification. It provides more information to perform error identification in exchange for a slight increase in testing time (approximately 1.46% longer for a test length of 100,000 test patterns). The second recommendation dispenses with intermediate signature testing and constructs the space compaction sequences for the entire test. Errors in the space compaction sequences can be used to target data recovery on specific areas of the test set. Diagnosis information is thus obtained during normal "pass/fail" testing rather than requiring a second test and comparing intermediate signatures. The attractive features of this approach, including a simplified testing process and reduced tester usage, make this the most practical proposed data recovery method.

The primary data recovery scheme is based on the modelling of parallel data compaction in [23]. The data construction and error identification algorithm for one erroneous data stream were originally developed by Dr. Sun et al. [15, 17]. The two alternative data recovery schemes were suggested by Dr. Sun during the course of this thesis research. The main contributions of this author are: (1) the development and

implementation of the error identification algorithms used in the three data recovery schemes; (2) the implementation of the software package, DR; and (3) the validation of the proposed data recovery schemes by means of computer simulation.

Extensive computer simulation results demonstrate the merits and feasibility of the proposed schemes. The data recovery schemes are tested with large pseudorandomly generated response vectors to gauge performance on realistically large problems. Further, the primary data recovery scheme is applied to faulty responses generated by fault simulation of ISCAS85 benchmark circuits [7] to demonstrate the feasibility of structural analysis and fault simulation to reduce the number of recovered solutions.

The following conclusions are reached from the simulation results:

(1) A failing block of response vectors (block = 256 vectors) can be recovered in approximately two minutes on a Sun Ultra 1 workstation. This was the maximum time required for the largest response vector size tested (1024×16 bits). Data recovery of this response size was performed based on 64 Kbytes of information transferred from the CUT, whereas the data retrieval method in [20] would transfer 512 Kbytes of data, an 8-fold saving. Fault diagnosis can be accomplished with only a few failing blocks of vectors [20]. Thus the use of expensive testing equipment can be minimized in exchange for several minutes of CPU time on less expensive workstations.

(2) The error identification algorithms may produce multiple recovered solutions. On average, approximately 20 solutions are produced when few response errors occur. The number of recovered solutions decreases as the percent errors and the number of failing vectors increase. This is analogous to fault diagnosis where faults that produce few errors are more difficult to diagnose than those that cause gross failure.

(3) Post-data recovery structural analysis and fault simulation are feasible to reduce the number of recovered solutions. Multiple responses recovered from benchmark circuits were successfully resolved to a single solution with subsequent structural analysis and fault simulation. Depending on the benchmark circuit, an average of 5 recovered solutions were reduced to a single solution after fault simulation. Structural analysis and fault simulation are already integral parts of fault diagnosis: the only additional expense for data recovery is the extra computation time needed to consider multiple recovered solutions.

The outcome of this research suggests the feasibility of the proposed data recovery schemes for VLSI circuits with STUMPS architectures. We are able to recover the lost circuit responses with a reduced amount of information transferred from the CUT to the tester which translates to shorter testing times for data retrieval, decreased tester usage and ultimately, significant savings in the overall testing and production cost.

The following problems have been identified and merit further investigation:

(1) The error identification algorithms are currently limited to identifying errors occurring in at most two scan chains. Further research is necessary to expand the applicability of the algorithms to multiple scan chains. The original scheme in [15] used the intersections of space compaction columns to identify circuit response errors, perhaps a variant of this method can be used to expand the error identification algorithms.

(2) BIST resources are inherently limited as designers are loath to surrender silicon area and I/O pins. The new data recovery scheme demands an additional scan-out port for the second MISR. This may be difficult to accommodate in some IC packaging.

Bibliography

[1] Miron Abramovici and Melvin A. Breuer. "Multiple Fault Diagnosis in Combinational Circuits Based on an Effect-Cause Analysis". *IEEE Transactions on Computers*, 29(6):451–460, 1980.

[2] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, 1990.

[3] Kendall E. Atkinson. *An Introduction to Numerical Analysis.* John Wiley & Sons, Inc., second edition, 1989.

[4] Sara Baase. *Computer Algorithms Introduction to Design and Analysis.* Addison Wesley, second edition, 1988.

[5] P. H. Bardell and W. H. McAnney. "Self-Testing of Multichip Logic Modules". In *Proceedings of the IEEE International Test Conference*, pages 200–203, 1982.

[6] Paul H. Bardell, William H. McAnney, and Jacob Savir. *Built-In Test For VLSI: Pseudorandom Techniques.* John Wiley & Sons, Inc., 1987.

[7] F. Brglez and H. Fujiwara. "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN". In *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1985.

[8] J. C. Chan and J. A. Abraham. "A Study of Faulty Signatures using a Matrix Formulation". In *Proceedings of the IEEE International Test Conference*, pages 553–561, 1990.

[9] Serge Demidenko, Vincenzo Diuri, and Alexander Ivaniukovich. "Error Localization in Test Outputs: a Generalized Analysis of Signature Compression". *Journal of Microelectronic Systems Integration*, pages 317–322, 1993.

[10] Shu Lin and Jr. Daniel J. Costello. *Error Control Coding: Fundamental and Applications.* Prentice-Hall, 1983.

[11] W. H. McAnney and J. Savir. "There is Information in Faulty Signatures". In *Proceedings of the IEEE International Test Conference*, pages 630–636, 1987.

[12] Benoit Nadeau-Dostie, Dwayne Burek, and Abu S. M. Hassan. "ScanBist: A Multifrequency Scan-Based BIST Method". *IEEE Design & Test of Computers*, pages 7–16, Spring 1994.

[13] J. Savir and W. H. McAnney. "Identification of Failing tests with Cycling Registers". In *Proceedings of the IEEE International Test Conference*, pages 322–328, 1988.

[14] H. S. Stone. *Discrete Mathematical Structures and their Applications.* Science Research Associates, Inc., 1973.

[15] X. Sun and W. Tutak. "Error Identification and Data Retrieval in Signature Analysis-based Data Compaction". In *Proceedings of the IEEE International Symposium on Defect & Fault Tolerance*, pages 177–184, 1996.

[16] X. Sun and W. Tutak. "Error Identification and Data Recovery in MISR-based Data Compaction". To appear in *Proceedings of the IEEE International Symposium on Defect & Fault Tolerance*, 1997.

[17] X. Sun and D. Yeung. "A Signature Analysis-based Fault Diagnosis Scheme for STUMPS"". Technical report, Department of Electrical and Computer Engineering, Univeristy of Alberta, 1995.

[18] Reginald P. Tewarson. *Sparse Matrices.* Academic Press, 1973.

[19] J. A. Waicukauski, V. P. Gupta, and S. T. Patel. "Diagnosis of BIST Failures by PPSFP Simulation". In *Proceedings of the IEEE International Test Conference*, pages 480–484, 1987.

[20] J. A. Waicukauski and E. Lindbloom. "Failure Diagnosis of Structured VLSI". *IEEE Design & Test of Computers*, 6(4):49–60, August 1989.

[21] Yuejian Wu. *Northern Telecom Diagnosis Scheme.* Personal communication, 1996.

[22] Yuejian Wu and Saman Adham. "BIST Fault Diagnosis in Scan-Based VLSI Environments". In *Proceedings of the IEEE International Test Conference*, 1996.

[23] Y. Zorian and A. Ivanov. "EEODM: an Effective BIST Scheme for ROMs". In *Proceedings of the IEEE International Test Conference*, pages 871–879, 1990.

Appendix A

DR User Reference Manual

Wes A. Tutak

Department of Electrical and Computer Engineering, University of Alberta

NAME

DataRecovery – A data recovery simulator.

SYNOPSIS

dr [options...] <config_file | S T V>

DESCRIPTION

This program implements the data recovery scheme presented in the author's M.Sc. thesis (August 1997). It is able to perform two types of data recovery simulations. The first is data recovery using faulty responses obtained from fault simulation of stuck-at faults in ISCAS85 benchmark circuits. And the second is data recovery from pseudorandomly generated response vectors.

The first simulation requires the name of a configuration file as the sole parameter. The contents of the configuration file are the following variables:

datadir	Specifies the directory containing the subsequent data files.
circuit	Specifies the name of an ISCAS85 circuit file.
dictionary	Specifies the name of the intermediate signature dictionary file.
faults	Specifies the name of the file containing the list of faults.

m	Determines the number of scan chains to partition the circuit outputs.
n	Determines the length of each scan chain.

The second simulations requires three parameters: S, T, V.

S	Specifies the number of scan chains containing errors.
T	Specifies the maximum number of errors per vector.
V	Specifies the number of failing vectors per block of 256 vectors.

OPTIONS

1. TYPE OF DATA RECOVERY SIMULATION

-iscas	Data recovery in ISCAS85 benchmark circuits (default).
-pseudo	Data recovery in pseudorandomly generated response vectors.

2. RECOVERY ALGORITHM OPTIONS

-n N	Set the length of scan chains (default N = 128).
-m M	Set the number of scan chains (default M = 16).
-b blocks	Consider the specified number of failing blocks (default blocks = 1).
-u	Use $Identify2Unknown()$ algorithm for error identification (default).
-k	Use $Identify2Known()$ algorithm for error identification.
-ov	Use overlapping response vectors (default).
-nov	Use non-overlapping response vectors.

3. OTHER FUNCTIONS

-dict n	Create an intermediate signature dictionary of length n.
-collapse	Perform fault collapsing on the given circuit.
-p	Print the internal circuit nodes and gates of the ISCAS85 benchmark circuit.
-sa	Compute the affected scan chain for each fault.

-oa Compute the affected output for each fault.

4. MISCELLANEOUS OPTIONS

-h Print out help summary.

-l Print the fault list.

-max mb Set the maximum number of blocks to consider during simulation (default $mb = 256$).

-r Use *time*() system call to seed random number generator.

-v Print our progress messages during execution.

-s seed Set *seed* value of random number generator.